Escola d'Enginyeria de Telecomunicació i Aeroespacial de Castelldefels
UNIVERSITAT POLITÈCNICA DE CATALUNYA

TRABAJO DE FIN DE CARRERA

TÍTULO DEL TFC: Estudio e implementación de métodos de protección de flujos audiovisuales en DVB-IP: AL-FEC y retransmisiones RTP

TITULACIÓN: Ingeniería Técnica de Telecomunicación, especialidad Telemática

AUTOR: Daniel Muñoz Cardete

DIRECTOR: David Rincón

DATA: 15 de noviembre de 2012

Título: Estudio e implementación de métodos de protección de flujos audiovisuales en DVB-IP: AL-FEC y retransmisiones RTP

Autor: Daniel Muñoz Cardete

Director: David Rincón

Data: 15 de noviembre de 2012

Resumen

DVB-IP es un estándar abierto desarrollado por el consorcio DVB (Digital Video Broadcasting) para la transmisión de contenidos multimedia sobre redes IP bidireccionales de banda ancha. El objetivo es la estandarización de estos servicios de transmisión de TV digital por Internet.

En este proyecto se han implementado dos mecanismos de protección contra pérdidas de paquetes en flujos de video en un escenario DVB-IP. Los dos modelos de protección implementados son las retransmisiones RTP y AL-FEC (Application Layer –Forward Error Correction). Todo se ha llevado a cabo con las especificaciones que la IETF y DVB dictan para la estandarización de este tipo de escenario de TV digital.

El desarrollo de las aplicaciones se ha realizado con el lenguaje de programación C. Hemos diseñado un escenario completo de streaming utilizando aplicaciones libres como son, VLC, Dvblast, LiveMediaServer555 y VLMA, y se ha implementado un servidor y un cliente de retransmisiones RTP, más una ampliación de un servidor de AL-FEC diseñado en un TFC previo. Para realizar las pruebas de funcionamiento hemos diseñado un servidor secundario que emula la pérdida de paquetes en el escenario de red.

También se ha realizado un estudio sobre el protocolo RTSP (Real Time Streaming Protocol) encargado de los diálogos entre servidores y clientes de streaming, de cara al desarrollo futuro de un cliente RTSP que siga las especificaciones DVB-IP. Se han utilizado diferentes aplicaciones de prueba incluidas en el software de Live555, para realizar diálogos entre un servidor y un cliente. Las pruebas realizadas nos han servido para averiguar el tipo de métodos y cabeceras que se utilizan en una aplicación RTSP y compararlo con el estándar y la organización DVB.

Title: Study and implementation of protection methods for audiovisual flows in DVB-IP: AL-FEC and RTP retransmissions

Author: Daniel Muñoz Cardete

Director: David Rincón

Date: Nov, 15th 2012

Overview

DVB-IP is an open standard developed by the DVB (Digital Video Broadcasting) consortium for the transmission of multimedia content over bidirectional broadband IP networks. The aim is to standardize these Digital TV broadcasting services over the Internet.

In this project we have implemented two different protection mechanisms for the protection of video streaming packets in a DVB-IP scenario. The two implemented protection models are RTP retransmissions and AL-FEC (Application Layer-Forward Error Correction). Our implementation has followed the IETF and DVB specifications that standardize this digital-TV-over-IP scenario.

The development has been done in C programming language. We have designed a complete scenario of video streaming using free applications such as VLC, Dvblast, LiveMediaServer555 and VLMa. We have implemented a server and a RTP retransmission client, and an extension of an AL-FEC server designed in a previous thesis. To perform the application tests we have designed a secondary server that emulates packet losses in a network scenario.

We have also carried out a study about RTSP (Real Time Streaming Protocol), the protocol responsible for the dialogues between servers and streaming clients. This study will be useful for a future RTSP client development compliant the DVB-IP specifications. We have used different test applications included in the Live555 software, for the dialogues between a server and a client. Tests helped us to identify the type of methods and headers used in an RTSP application and to compare it with the IETF standard and the DVB specifications.

ÍNDICE

INTRODUCCIÓN

Desde la invención de la televisión a principios del siglo XX, ésta se ha visto en constante evolución tecnológica. En la última década hemos podido ver el salto a la televisión digital, hemos estado presentes en las primeras emisiones de televisión en tres dimensiones, y ha llegado incluso a nuestros ordenadores personales y terminales móviles. En resumidas cuentas, no ha parado de avanzar des de el día de su invención.

Nosotros nos vamos a centrar en IPTV, la televisión que se transmite a través de la red de redes (Internet) y de la estandarización de éste servicio. Hoy en día las grandes operadoras de comunicaciones han desarrollado en mayor o menor medida servicios de televisión por IP, como puede ser Movistar con Imagenio [39], Orange con Orange TV [40] o Jazztel con Yomvi [41], por mencionar algunos casos. Los servicios que nos ofrecen estos proveedores son televisión por IP, servicios interactivos, videos bajo demanda, etc. El problema de estos servicios es que no utilizan un estándar para la emisión sino que cada empresa se basa en tecnologías propietarias, lo cual condiciona el uso de los servicios y repercute en el coste de éstos.

El consorcio DVB (Digital Video Broadcasting) [34] se desarrolló para estandarizar la transmisión de TV digital sobre redes terrestres, satélite o cable, y define aspectos que abarcan desde la capa más física, donde se describen las modulaciones a utilizar para los diferentes medios de transmisión (DVB-T terrestre, DVB-S satélite, DVB-C cable), hasta la señalización empleada para la sintonización y reproducción del flujo de video, pasando por aspectos de sincronización y encriptación.

A partir de esta situación aparece en escena DVB-IP [2], un estándar desarrollado por el consorcio DVB para el envío de televisión digital sobre el protocolo IP a través de redes de acceso de banda ancha (la más común es ADSL, pero también se pueden usar otras redes de acceso).

Una de las grandes ventajas de la estandarización ofrecida por DVB-IP es el abaratamiento de los costes de producción de los equipos, pudiendo realizar grandes producciones para un amplio mercado internacional. Además, al tratarse de un estándar abierto se reducen los posibles costes del uso de tecnologías privadas. El usuario final podrá beneficiarse de la utilización de equipos que cumplieran el estándar sin tener que preocuparse de su ISP (Internet Service Provider), pudiendo seleccionar el proveedor de DVB-IP a visualizar.

DVB especifica todos los protocolos y mecanismos que deberá cumplir el cliente DVB-IP y especifica varios tipos de servicios de IPTV, como por ejemplo "Live Media" o “CoD” (Content on Demand). Para operadores o fabricantes que no requieran una oferta completa de todas las funciones que se ofrecen en DVB-IP, se introduce el concepto de perfil. Estos perfiles se definen como niveles de funcionalidad, para que terminales con más o menos complejidad puedan interactuar con los servidores de DVB.

En el presente Trabajo Fin de Carrera se ha realizado un estudio de los servicios y funcionalidades que proporciona DVB-IP, centrándonos en la parte que refiere a la protección de los flujos multimedia a partir de mecanismos de retransmisión RTP y AL-FEC (Application Layer- Forward Error Correction). Se ha diseñado un servidor y un cliente de retransmisiones que se adecua a lo que marca el estándar [2], y se ha modificado un servidor AL-FEC que diseñó un estudiante anterior [1] con el objetivo de mejorar la protección del servicio DVB-IP.

También se ha realizado un estudio del protocolo RTSP (Real Time Streaming Protocol). Dicho protocolo es utilizado básicamente en entornos de streaming, ya que es capaz de iniciar y mantener una conexión entre un cliente y un servidor. En nuestro escenario RTSP se utiliza para realizar el video bajo demanda, y el estudio que presentamos prepara el terreno para un futuro TFC centrado en el desarrollo de un servidor y cliente RTSP adaptados a los requisitos de DVB-IP.

A continuación mostramos un resumen de los diferentes capítulos que forman el trabajo.

- Capítulo 1: En este capítulo se presentan las principales características de DVB-IP, el escenario donde debe desarrollarse, los diferentes perfiles que ofrece DVB-IP y los diferentes protocolos que participan en el desarrollo de los servicios DVB-IP.

- Capítulo 2: En este capítulo nos centramos en el descubrimiento de servicios DVB-IP, detallando los pasos a seguir para que un usuario pueda visualizar la oferta de servicios.

- Capítulo 3: En este capítulo se estudia el protocolo RTSP, utilizado para la transmisión de video bajo demanda por parte del usuario.

- Capítulo 4: En este capítulo se presenta el reto de la protección de los datos de video, planteando las dos posibilidades que ofrece DVB-IP: retransmisiones RTP y AL-FEC.

- Capítulo 5: En este capítulo mostramos las aplicaciones desarrolladas: el diseño de un servidor de retransmisiones RTP y un cliente, y una mejora de un servidor de AL-FEC diseñado en un TFC anterior, así como software auxiliar para emular las pérdidas de paquetes en nuestro escenario de pruebas. También se valoran los resultados obtenidos.

CAPÍTULO 1. Introducción a DVB-IP

1.1. Introducción a TV por IP y DVB-IP

En muchos países se ha desarrollado en mayor o menor medida servicios de IPTV. El aumento del ancho de banda de las redes IP y la aparición de nuevas tecnologías de acceso (como ADSL, por ejemplo) han permitido el despliegue de servicios de TV, que hasta ahora estaban reservados para otros medios, como por ejemplo el cable o el satélite. La definición de IPTV (Internet Protocol Televisión) hace referencia únicamente al mecanismo de transmisión del servicio de televisión (servicios de alta calidad, comparable a la TV actual analógica o digital) a través de redes IP.

El término IPTV (a veces denominado IPTV *over managed networks* [31]) suele referirse a un entorno cerrado en el que el proveedor del servicio controla tanto la red de transmisión como los contenidos o el acceso a los mismos. Un ejemplo sería Imagenio, en el que Telefónica controla tanto la red de transporte como la red de acceso al usuario. También existe lo que se denomina Internet TV o servicios *over-the-top* (OTT) [31], el cual esta basado en transmisiones de video en streaming utilizando la Internet global para llegar a los usuarios. Un ejemplo de esta modalidad sería YouTube [42]. La televisión por Internet representa un entorno menos controlado, en el que tanto los contenidos como su acceso tienen un carácter más abierto. En contraposición tiene una baja calidad de visualización y no hay ningún tipo de garantías de calidad del servicio (por ejemplo, la reproducción puede pararse o tener menor calidad por congestión en la red).

DVB-IP es el estándar desarrollado por el consorcio DVB para el escenario IPTV. Ofrece un servicio de envío de televisión digital a través de redes de banda ancha y define las especificaciones para el descubrimiento de servicios, la difusión, transmisión y protección de los flujos de video.

Las especificaciones que definen los aspectos tratados en este TFC están publicadas en los documentos siguientes:

- Digital Video Broadcasting (DVB); Transport of MPEG-2 TS Based DVB Services over IP Based Networks. ETSI, DVB Bluebook A86 [2].

- Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IP Phase 1 specifications, ETSI TS 102 542 V1.2.1. [3].

- Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IP Phase 1 specifications. Part 1: Core IPTV Functions TS 102 542-1 V1.3.2 [4].

- Digital Video Broadcasting (DVB); Specification for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream, ETSI TS 101 154 V1.10.1 (2007-07) [5].

1.2. Arquitectura DVB-IP

La arquitectura básica de un sistema DVB-IP está formada por un Proveedor de Contenidos, el Proveedor de Servicios, el Home Network Gateway y el HNED (Home Network End Device). En la figura 1.1 podemos observar el esquema básico de un escenario DVB-IP.

Proveedor de Contenidos: Se refiere a la entidad que posee contenidos o tiene licencia para venderlos. Por ejemplo podría ser Telecinco o La Sexta, que deciden ofrecer sus contenidos vía Internet.

Proveedor de Servicios: Es la entidad que proporciona el servicio al usuario final. Existen diferentes tipos de proveedor de servicios para DVB-IP, un ejemplo podría ser directamente el ISP (Internet Service Provider), como es el caso de Movistar en su servicio Imagenio.

Home Network Gateway: Es el dispositivo encargado de interconectar la red del Proveedor de servicios con la red local del usuario, por ejemplo el router local.

Home Network End Device (HNED): Es el dispositivo conectado a la red local instalado en casa del usuario y donde generalmente termina el flujo de datos IP, aunque no necesariamente ha de ser el equipo final de los flujos de datos que no sean IP, ya que este equipo podría hacer una retransmisión de estos datos recibidos hacia otra tecnología que no fuera IP (por ejemplo hacia un monitor de TV). Típicamente será un decodificador que puede ser independiente o bien estar integrado en el router ADSL, pero también podría estar integrado en una Smart TV que tenga conexión Ethernet.

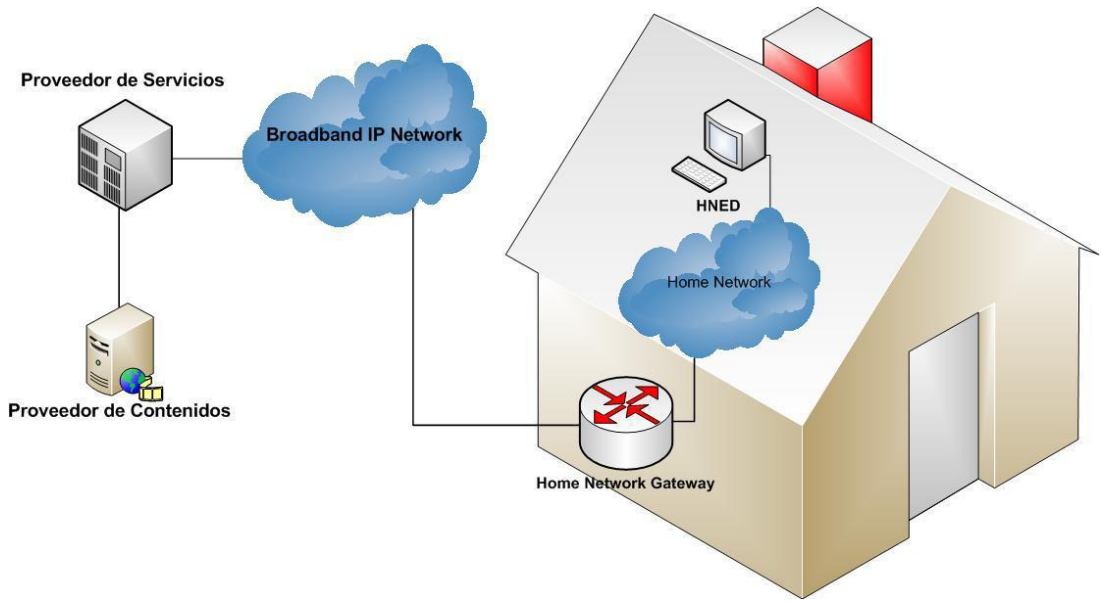

Fig. 1.1. Esquema básico de un escenario DVB-IP.

1.3. Perfiles DVB-IP

Para operadores o fabricantes que no requieran un completo uso de todas las funciones que se ofrecen en DVB-IP, se introduce el concepto de perfil. Estos perfiles se definen como niveles de funcionalidad, para que terminales con más o menos complejidad puedan interactuar con los servidores DVB-IP.

Cada uno de los perfiles acoge un conjunto de servicios diferenciados que trabajan sobre protocolos diferentes. DVB especifica los perfiles en su estándar [10].

1.3.1. Basic profile

Este perfil solo trabaja con UDP (User Datagram Protocol) a nivel de transporte, siendo su entorno de aplicación escenarios controlados (como por ejemplo una red Local) en los cuales no sea necesario el uso de RTP (Real Time Protocol) ni RTCP (Real Time Control Protocol). Los contenidos que se pueden visualizar son canales Live Media, es decir canales emitidos vía multicast, ya que tan solo incluye el protocolo IGMP (para unirse y mantener la conexión en grupos multicast; se describirá más adelante). Los contenidos únicamente podrán estar codificados en MPEG-2.

1.3.2. Live Media profile

Este perfil, además de las funcionalidades del perfil *Basic*, implementa la opción de trabajar con RTP/UDP, siendo posible su entorno de aplicación escenarios mas abiertos (como por ejemplo Internet a través de ADSL) en los cuales no se tenga tanto control sobre la calidad de la transmisión, ya que dispone de RTP y RTCP para monitorizar el estado de la conexión. La codificación de los videos podrá ser cualquiera de las que se indican en el estándar [5], incluyendo H.264/AVC.

1.3.3. Content on Demand (CoD) profile

Este perfil es el más completo de todos y el que cumple con todos los servicios posibles en DVB-IP, ya que incluye los dos anteriores y añade funcionalidades extra. Se pueden visualizar tanto canales Live Media, como Live Media con *Trick Mode* (de similares características a los Live Media pero con las opciones de Play, Stop, Forward, etc.) o servicios de Content on Demand. Para ello incluye el uso del protocolo de señalización RTSP (Real Time Streaming Protocol). Para la información de metadatos sobre los programas se podrá visualizar la EPG (Electronic Program Guide) ya que contempla el funcionamiento de BCG (Broadband Content Guide), contemplada en el estándar [6].

1.3.4. Content Dowload Service (CDS)

Este perfil permite la descarga de contenidos a un dispositivo de almacenamiento local en el HNED a través de una conexión IP de banda ancha. Está pensado para proveer de servicios IPTV áreas en las que la conexión no es adecuada para servicios de streaming o es propensa a errores, así como para ofrecer un servicio adicional de descarga de contenidos independiente del ancho de banda asignado a las transmisiones en directo o CoD, por lo que éstas no se vean afectadas con la distribución de este nuevo

servicio “en diferido”. Esto permitiría, por ejemplo, descargar durante la noche películas en el disco duro del decodificador del usuario, para poder ser visualizadas posteriormente sin estar necesariamente conectado a la red.

1.4. Pila de protocolos DVB-IP

DVB utiliza algunos protocolos comunes en las redes IP para señalizar, transportar y controlar las transmisiones de video y audio. A continuación mostramos un cuadro de los diferentes protocolos y una breve explicación de cada uno de ellos.

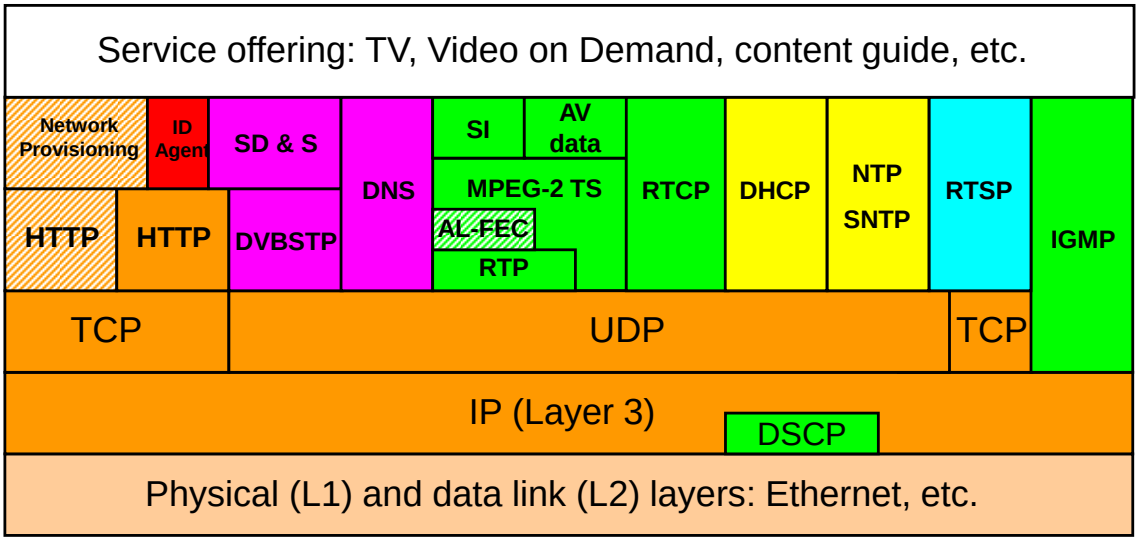

Fig. 1.2. Protocolos utilizados en DVB-IP [45]

1.4.1. IGMP

Internet Group Management Protocol (IGMP) es un protocolo que permite a un terminal señalizar a los routers IP multicast a qué grupos se quiere unir (o abandonar) el usuario. En DVB-IP se utiliza para unirse a los grupos multicast correspondientes a los servicios Live Media y mantener esta conexión a partir del envío de informes del estado de la pertenencia. IGMP es un pilar básico del estándar DVB-IP, ya que la difusión de los canales de TV convencional (los mismos que se ofrecen por TDT u otros medios) se realizan sobre multicast.

1.4.2. RTP

Real-time Transport Protocol (RTP) es un protocolo de nivel de sesión utilizado para la transmisión de información en tiempo real, como por ejemplo audio y vídeo. En DVB-IP se utiliza para el encapsulamiento de los datos audiovisuales y su posterior transmisión. La encapsulación se realiza de la siguiente manera: en cada paquete IP se introducirá un número entero de paquetes MPEG-2 Transport Stream (TS) [ref a wikipedia], de 188 bytes cada uno, por lo que tendremos un paquete como el mostrado en la Figura 2.1. Podemos ver como lo componen los paquetes MPEG-2 TS y las cabeceras IP, UDP y RTP.

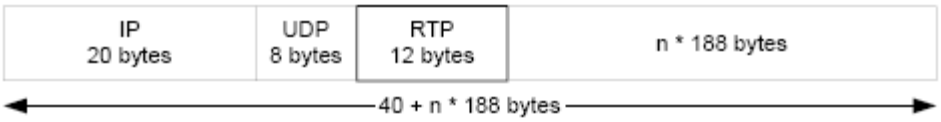

Fig. 1.3. Encapsulamiento de los datos audiovisuales sobre RTP [1].

1.4.3. RTCP

Real-time Transport Control Protocol (RTCP) es el protocolo que proporciona información de monitorización del flujo RTP asociado. Trabaja junto con RTP en el transporte, pero no transporta ningún dato por sí mismo, sino estadísticas de cómo va la transmisión, informando de la calidad de servicio de la transmisión RTP. También se usa habitualmente para transmitir paquetes de control a los participantes de una sesión multimedia de streaming.

1.4.4. DNS

Domain Name System (DNS) es el protocolo encargado de relacionar direcciones IP con nombres de dominio. En nuestro caso el servidor DNS es el que relacionará el proveedor de servicios DVB con el cliente, ya que este proveedor tendrá asignado un nombre único.

1.4.5. AL-FEC

Application Layer-Forward Error Correction (AL-FEC) es un mecanismo de corrección de errores que permite al usuario corregir errores sin utilizar retransmisiones. La corrección se basa en enviar un flujo de datos redundante al cliente que éste podrá utilizar en caso de perdida de paquetes. En este trabajo nos hemos centrado en éste tipo de protección para DVB. Más adelante se ofrece información más detallada sobre el mecanismo.

1.4.6. DHCP

Dynamic Host Configuration Protocol (DHCP) es el protocolo de red que se utiliza para asignar direcciones IP y otros parámetros de red a los equipos cuando se conectan a una red. En DVB-IP se utiliza para los equipos HNED.

1.4.7. NTP

Network Time Protocol (NTP) es un protocolo de Internet para sincronizar los relojes de los sistemas informáticos en redes con latencia variable. NTP utiliza UDP como su capa de transporte, usando el puerto 123. En DVB se utiliza para sincronizar los flujos de video y audio entre el servidor y los clientes.

1.4.8. DVBSTP

DVB Streaming Transport Protocol (DVBSTP) es un protocolo utilizado en su totalidad en DVB para el transporte de los registros XML en el descubrimiento de servicios cuando trabajamos con grupos multicast o con vídeo bajo demanda. Es ligero y se transporta sobre UDP. El protocolo hace especial mención a la fragmentación de los documentos, ya que éstos superan normalmente la MTU, por esta cuestión realizan un paso previo de segmentación, y en cada paquete tiene que haber un solo segmento XML.

1.4.9. HTTP

Hypertext Transfer Protocol (HTTP) es un protocolo genérico para la transferencia de páginas web en Internet, pero en DVB se utiliza para el descubrimiento de servicios cuando se trabaja con direcciones unicast. Se transporta sobre TCP y la fragmentación está controlada por éste.

CAPÍTULO 2. Descubrimiento de Servicios

2.1. Service Discovery & Selection (SD&S)

Service Discovery & Selection (SD&S) es un mecanismo que consiste en la presentación de una guía electrónica de programas DVB disponibles. Le permitirá al usuario disponer de todos los datos concernientes a los servicios disponibles, todos ellos en formato XML (a lo que denominaremos informe SD&S). En este informe el HNED podrá ver la ubicación del servidor, como contactar con él (dirección IP y puerto), el nombre de cada uno de los programas ofrecidos, su descripción, lenguaje de emisión, si dispone de teletexto, etc. Una vez el cliente se decida a recibir un programa SD&S le permitirá hacer dicha petición y recibir el flujo en su terminal (decodificador o TV. Disponemos de dos métodos de transporte de la información:

- Modo Pull: En este caso el terminal es el que inicia de manera activa la conexión con el servidor solicitando información de los servicios. Este método trabaja normalmente con una conexión unicast y sobre TCP

- Modo Push: En este caso no se establece una conexión con el servidor, sino que la información es emitida por el servidor de forma continua en un grupo multicast, y el cliente la recibe de forma pasiva suscribiéndose al grupo. Este modo trabaja sobre UDP.

A continuación en la imagen 2.1 mostramos los pasos a seguir para descubrir los servicios ofrecidos por un proveedor de servicios

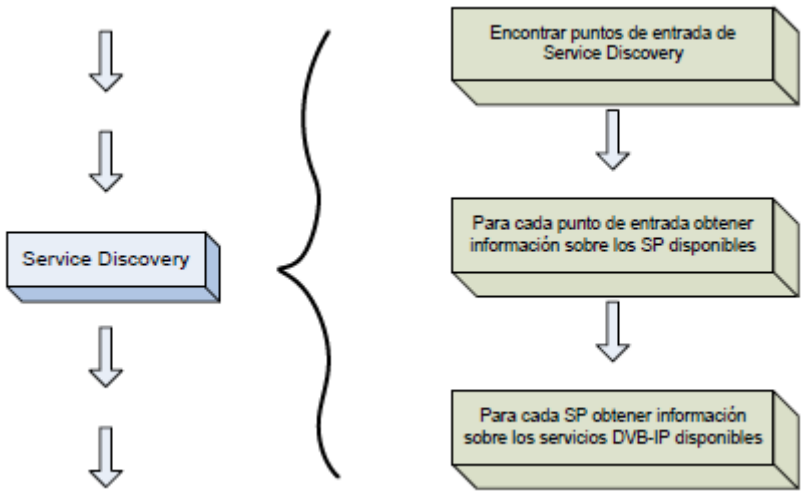

Fig 2.1 Pasos para descubrir los servicios DVB-IP [46].

A continuación analizaremos cada uno de los pasos a seguir para encontrar los servicios disponibles en una oferta de DVB-IP.

2.1.1. Paso 1: Encontrar los Puntos de entrada

El primer paso es encontrar los puntos de entrada SD&S, donde obtendremos la información sobre los distintos proveedores de servicios disponibles. El proceso de localización de punto de entrada se realiza de forma priorizada, empezando en la primera de las opciones disponibles que existe y saltando a la siguiente en el caso de no obtener éxito en el punto de entrada actual.

Los cuatro posibles puntos de entrada son los siguientes:

- Cuando el HNED consigue una dirección IP a partir de un servidor de DHCP, puede conseguir un nombre de dominio a partir de la opción 15 de DHCP. A partir de ahí se pueden conseguir varios puntos de entrada vía DNS de acuerdo con el RFC 2782 [7]. De no haber conseguido ningún nombre de dominio tendremos que pasar al siguiente método.

- Una dirección multicast registrada por IANA para tal propósito (DvbServDisc), con la dirección IP 224.0.23.14. En el caso de no recibir ningún paquete DVBSTP desde esta dirección IP multicast, se pasara al siguiente punto de entrada.

- Mediante DNS se puede adquirir una lista de puntos de entrada SD&S según el servicio de localización RFC 2782 [7]. El nombre del servicio es _dvbservdsc, el protocolo puede ser TCP o UDP, mientras que el resto del nombre es el nombre de dominio mantenido por DVB para el servicio de descubrimiento, este nombre de dominio está establecido en services.dvb.org.

  Los nombres son _dvbservdsc._tcp.services.dvb.org o bien _dvbservdsc._ udp.services.dvb.org. Esto requiere que el HNED soporte DNS SRV de acuerdo a la especificación en el RFC 2782 [7]. La organización DVB mantendrá el nombre de dominio para services.dvb.org y los nuevos proveedores de servicios deberán registrarse en DVB para añadirlos a la lista de DNS SRV. Estos servicios serán publicados mediante HTTP para el protocolo TCP, mientras que para el servicio multicast se publican a través del protocolo UDP. En el caso de no obtener ninguna respuesta ante la consulta DNS, se pasará al siguiente punto de entrada.

- Si no se ha encontrado ningún punto de entrada a través de los pasos anteriores, el usuario del HNED podrá introducir directamente los puntos de entrada, la dirección IP, como parte de los datos de la configuración del terminal.

2.1.2. Paso 2: Para cada punto entrada encontrar los Service Provider disponibles

Una vez tengamos un punto de entrada de información pasaremos a recibir informes en el cliente, detallando la información necesaria para que este pueda elegir un servicio. La primera elección deberá ser la de un *Service Provider* (SP) en concreto para descubrir los servicios que ofrece.

La “carta de presentación” de los *Services Providers* se traduce en un informe en formato XML (Service Provider DiscoveryInformation), que puede entregarse tanto en modo multicast, modelo *push*, como en modo unicast, modelo *pull*, o petición-respuesta. Los dos modelos están soportados.

2.1.3. Para cada Service Provider obtener los servicios DVB-IP disponibles

Una vez decidido el SP del que queremos obtener la información pasaremos a recibir los informes acerca de los servicios de que dispone. Para ello recibiremos informes DVB-IP Offering Record.

Un proveedor de servicios puede ofrecer dos tipos de servicios bien diferenciados, los Livemedia ("TS Full SI" o "TS SI Optional ") y/o Contenido bajo demanda (a través de la BCG Discovery Record).

Para la entrega de servicios Livemedia existen los modelos siguientes:

- *TS Full SI Broadcast Discovery* deriva de DVB-IP Offering Record, provee toda la información necesaria para encontrar los servicios Live Media disponibles que llevan integrados la información del SI (Service Information) en el Transport Stream. Las tablas SI describen el canal de TV, cuántos audios y teletextos / subtítulos están asociados, etc [ref Wikipedia tablas SI].

- *TS Optional SI Broadcast Discovery Information* implementa la misma información que el "TS full SI", además de los servicios de Service Description Location del SI. Básicamente contiene la misma información que el TS FULL SI pero con alguna información más sobre el contenido que se está enviando. Se emplea cuando no se está enviando la información del SI en el flujo Transport Stream

Para la entrega de contenido bajo demanda:

- *Broadband Content Guide (BCG) Discovery Record* proporciona una guía de contenidos, mediante la cual se pueda obtener información acerca de los diferentes servicios multimedia ofrecidos por DVB-IP

2.1.4. Identificación del servicio

Para que el descubrimiento de los servicios pueda funcionar deberemos poder identificar al SP y los servicios ofrecidos por este. El SP se identifica únicamente con el nombre del dominio DNS que se haya registrado. Para los servicios existen dos mecanismos básicos para su única identificación, de los que se usará la tripleta DVB. Se trata de un trío de identificadores numéricos: original_network_id, transport_stream_id y service_id, con lo cual permite distinguir entre el mismo servicio transportado por diferentes redes. Por ejemplo, el trío distinguiría TVE1 transportado por Hispasat o por Astra.

2.1.5. Selección del servicio

Se definen dos métodos bien diferenciados para el acceso a los servicios de stream desde el HNED:

- RTSP, para modo unicast

- IGMP, para modo multicast

Los servicios Live Media son transmitidos sobre IP multicast de forma continua, no necesitan ser iniciado por ningún HNED. Los dispositivos finales pueden solicitar el acceso o abandonar el acceso a un flujo de datos concreto mediante IGMP. El campo Service Location del paquete Broadcast Record proporciona la información necesaria para enviar los mensajes IGMP para cada servicio anunciado.

Opcionalmente los servicios de distribución de vídeo en tiempo real pueden elegir requerir al HNED que ejecute una serie de pasos, como activar una cuenta, acceso condicional (acceso solo a los canales que tenga el usuario contratado), etc. En estos sistemas se usará RTSP, que además servirá para adquirir parámetros requeridos por IGMP. Por ejemplo, para el acceso a canales de pago que estén siendo retransmitidos en multicast, usaremos RTSP para obtener la información necesaria en el uso de IGMP y la visualización de los canales multicast.

Los servicios Live Media con el Trick Mode son similares a los Live Media pero proporcionan la posibilidad al usuario de control (avance, stop, etc.) y serán entregados mediante IP unicast usando RTSP

CAPÍTULO 3. Estudio de RTSP

3.1. Protocolo

El protocolo RTSP (Real Time Streaming Protocol) es un protocolo de señalización para flujos multimedia en tiempo real, capaz de controlar y sincronizar datos, ya sean de audio o vídeo.

3.1.1. Descripción

El protocolo RTSP no está orientado a conexión; en su lugar, el servidor mantiene una sesión asociada a un identificador. En la mayoría de los casos RTSP usa TCP para los datos de control y UDP para los datos de audio y vídeo (aunque también es posible utilizar TCP para los datos de audio/vídeo). El cliente es capaz de abrir diferentes sesiones para poder satisfacer sus necesidades.

RTSP tiene una sintaxis parecida a HTTP pero difiere de este protocolo en lo siguiente:

- Introduce nuevos métodos y tiene un identificador de protocolo diferente.
- Un servidor RTSP necesita mantener el estado de la conexión, HTTP no.
- Tanto el servidor como el cliente pueden realizar peticiones, en HTTP solo el cliente.

RTSP únicamente señaliza; los datos se transportan en un protocolo diferente (típicamente UDP -con o sin RTP- o bien TCP).

3.1.2. Propiedades

A continuación mostramos las principales características del protocolo RTSP:

- El protocolo es extensible; se pueden añadir métodos y parámetros fácilmente.
- Puede ser seguro, reutilizando mecanismos de seguridad web, como la autenticación.
- Capacidad multi-servidor. Cada flujo multimedia dentro de una presentación puede venir de diferentes servidores. El cliente establece varias sesiones y la sincronización se realiza en la capa de transporte.
- Es adecuado para aplicaciones profesionales, ya que soporta resolución a nivel de frame mediante marcas temporales SMPTE.

3.2. Métodos

Las peticiones están basadas en peticiones HTTP y generalmente se envían de cliente a servidor.

Announce

Se envía des del cliente al servidor o viceversa para enviar la descripción de una presentación identificada por un URL. Cuando se envía desde el servidor al cliente es para actualizar la descripción de sesión en tiempo real.

Setup

El cliente le pide al servidor que reserve recursos para un stream y para iniciar una sesión RSTP. Si el cliente envía una petición de SETUP al servidor mientras este está reproduciendo el medio, si éste no soporta o permite este tipo de caso, debe devolver un error, específicamente el 455.

Options

El comportamiento es similar al del protocolo HTTP. El cliente y el servidor le comunican a la otra parte las opciones que pueden aceptar.

Describe

Este método obtiene una descripción del fichero o flujo multimedia apuntado por una URL RTSP situada en un servidor. El servidor responde a esta petición con una descripción del recurso solicitado, y entre otros datos la descripción contiene una lista de los flujos multimedia que serán necesarios para la reproducción. Esta solicitud/respuesta constituye la fase de inicialización del RTSP.

Play

Una petición de PLAY provocará que el servidor comience a enviar datos de los flujos especificados utilizando los puertos configurados con el SETUP.

Pause

Detiene temporalmente uno o todos los flujos, de manera que puedan ser recuperados con un PLAY posteriormente.

Record

El cliente inicia la grabación de unos datos multimedia de acuerdo a la descripción de la presentación. El servidor decide si desea almacenar la grabación bajo un Request-URI o en otro URI.

Get Parameter

Recupera el valor de un parámetro de una presentación o un stream identificado por su URI.

Set Parameter

Asigna o cambia el valor de un parámetro de una presentación o de un stream identificado por su URI. Una petición debería tener pocos parámetros (preferiblemente uno) para así permitir al cliente determinar la causa de un

posible error en el request. Si el request contiene muchos parámetros, el servidor debería actuar solo si todos son correctos. Es por ello que un servidor debe permitir cambiar el valor de dichos parámetros rápidamente.

Redirect

El servidor informa al cliente que debe conectarse a otra dirección de servidor. La cabecera obligatoria de localización indica el URL al cual el cliente se debe conectar.

Teardown

Detiene la entrega de datos para la URL indicada liberando los recursos asociados.

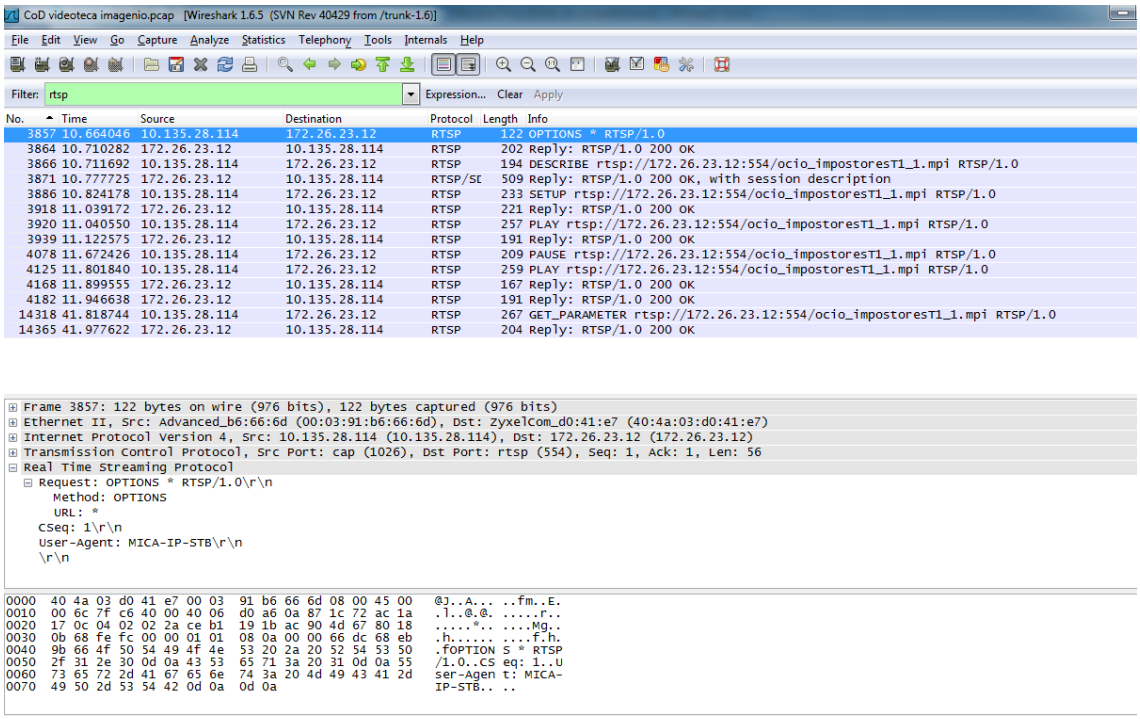

Fig 3.1 Captura de Imagenio [39], adquiriendo video bajo demanda.

En la figura 3.1 se muestra un dialogo entre un cliente y un servidor de Imagenio que utiliza RTSP para gestionar el video bajo demanda. La prueba fue realizada accediendo al videoclub de Imagenio [39] y seleccionando videos gratuitos de una serie de TV.

A continuación vamos a esquematizar y a explicar detenidamente todos los mensajes que se enviaron entre el cliente y el servidor. En la figura 3.2 podemos observar como se ha realizado la petición de un cliente para visualizar un video bajo demanda del servidor de Imagenio [39].

a) El cliente comienza haciendo una petición de OPTIONS al servidor, el cual responde con un 200 OK, y mostrando las opciones de las que dispone, en este caso son: Options, describe, setup, teardown, play,

pause, get parameter, set parameter. Así el servidor le indica al cliente lo que es capaz de entender.

b) El cliente, cuando recibe estos datos, le envía un DESCRIBE. Esta petición se utiliza para recibir información sobre una URL RTSP (el video que queremos ver) y el servidor contesta con un 200 OK adjuntando la descripción de la sesión.

c) En éste momento el cliente ya tiene los datos de la URL que quiere visualizar y envía un SETUP al servidor para que reserve recursos para la reproducción del medio que queremos. El servidor contesta con 200 OK indicándole al cliente el número de sesión y los datos de transporte (donde se indica la dirección IP y el puerto).

d) A partir de aquí el cliente tiene toda la información pertinente al video que quiere reproducir y envía un PLAY al servidor para que empiece la transmisión del video.

e) La siguiente prueba es pausar la reproducción y volver a reproducirla, enviando al servidor una petición de PAUSE y a continuación otro PLAY.

f) En un cierto puntual el cliente le envía un GET-PARAMETER al servidor con intención de recuperar información sobre el medio que estamos reproduciendo, con la intención de mantener la transmisión que estamos realizando, el servidor contesta con un 200 OK y especificando algunos de los parámetros de la conexión.

g) Para finalizar la transmisión es necesario que el cliente envíe un TEARDOWN al servidor para que éste deje de transmitir el flujo multimedia y así liberar los recursos.

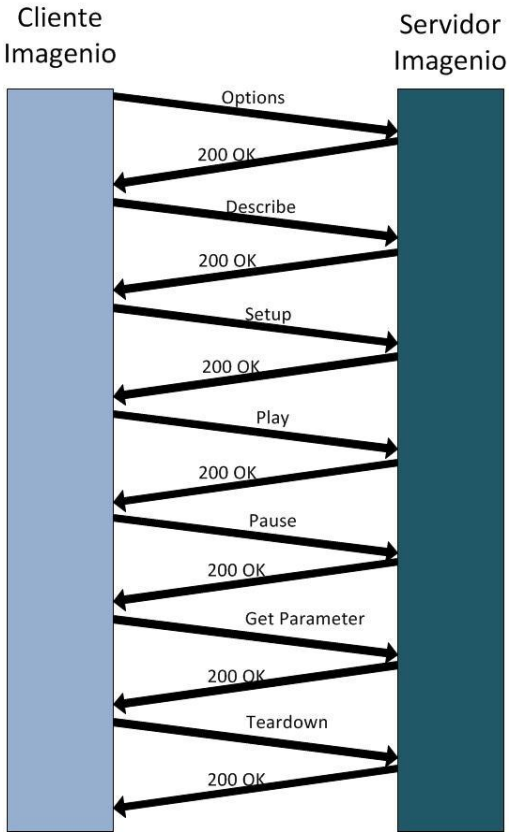

Fig 3.2 Dialogo RTSP entre el cliente y el servidor de Imagenio.

3.3. Comparación IETF/DVB/Live555

A continuación se muestra una tabla donde se compara lo que el estándar RFC 2326 [8] y la organización DVB [9] marcan para la utilización de los métodos en un diálogo RTSP entre un cliente y un servidor de vídeo, con la aplicación que hemos utilizado para realizar pruebas de CoD (Live555 [25]).

Tabla 3.1 Comparativa de importancia de los métodos en RTSP según IETF y DVB, y uso en la aplicación LiveMediaServer555

RTSP Method	Direction (H = HNED, S = SERVER)	IETF	DVB Requirement	Live555 MediaServer
ANNOUNCE	H→S	MAY	MAY	OK
ANNOUNCE	S→H	MAY	SHOULD	KO
DESCRIBE	H→S	SHOULD	SHOULD	OK
GET_PARAMETER	H→S	MAY	SHOULD	OK
GET_PARAMETER	S→H	MAY	MAY	OK
OPTIONS	H→S	SHALL	SHALL	OK
OPTIONS	S→H	MAY	MAY	OK
PAUSE	H→S	SHOULD	SHALL	OK
PLAY	H→S	SHALL	SHALL	OK
REDIRECT	S→H	MAY	SHALL	KO
SETUP	H→S	SHALL	SHALL	OK
TEARDOWN	H→S	SHALL	SHALL	OK

Para la comprobación de estos mensajes se ha hecho un estudio del código de la aplicación Live555MediaServer, y luego se han realizado pruebas con una aplicación de pruebas que viene en el código fuente de Live555, con la cual se puede simular un cliente RTSP y ver el dialogo entre éste servidor de vídeos bajo demanda y un cliente. Se muestra a continuación un ejemplo de dialogo completo.

```
dani@ubuntu:~/Descargas/live/liveMedia$ openRTSP rtsp://192.168.1.33:8554/para.m
4e
Opening connection to 192.168.1.33, port 8554...
...remote connection opened
Sending request: OPTIONS rtsp://192.168.1.33:8554/para.m4e RTSP/1.0
CSeq: 2
User-Agent: openRTSP (LIVE555 Streaming Media v2011.07.21)


Received 152 new bytes of response data.
Received a complete OPTIONS response:
RTSP/1.0 200 OK
CSeq: 2
Date: Tue, Sep 04 2012 20:49:12 GMT
Public: OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, GET_PARAMETER, SET_PARA
METER
```

Fig 3.3 Método Options.

El cliente inicia el dialogo enviando un OPTIONS al servidor para que éste le ofrezca todas las posibilidades que puede ofrecer. En este caso el servidor le indica que entiende OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE, GET-PARAMETER y SET-PARAMETER.

```
Sending request: DESCRIBE rtsp://192.168.1.33:8554/para.m4e RTSP/1.0
CSeq: 3
User-Agent: openRTSP (LIVE555 Streaming Media v2011.07.21)
Accept: application/sdp


Received 665 new bytes of response data.
Received a complete DESCRIBE response:
RTSP/1.0 200 OK
CSeq: 3
Date: Tue, Sep 04 2012 20:53:15 GMT
Content-Base: rtsp://192.168.1.33:8554/para.m4e/
Content-Type: application/sdp
Content-Length: 498

v=0
o=- 1346791752965357 1 IN IP4 192.168.1.33
s=MPEG-4 Video, streamed by the LIVE555 Media Server
i=para.m4e
t=0 0
a=tool:LIVE555 Streaming Media v2011.11.20
a=type:broadcast
a=control:*
a=range:npt=0-
a=x-qt-text-nam:MPEG-4 Video, streamed by the LIVE555 Media Server
a=x-qt-text-inf:para.m4e
m=video 0 RTP/AVP 96
c=IN IP4 0.0.0.0
b=AS:500
a=rtpmap:96 MP4V-ES/90000
a=fmtp:96 profile-level-id=3;config=000001B003000001B509000001000000012000C88880
07D05841214103
a=control:track1

Opened URL "rtsp://192.168.1.33:8554/para.m4e", returning a SDP description:
v=0
o=- 1346791752965357 1 IN IP4 192.168.1.33
s=MPEG-4 Video, streamed by the LIVE555 Media Server
i=para.m4e
t=0 0
a=tool:LIVE555 Streaming Media v2011.11.20
a=type:broadcast
a=control:*
a=range:npt=0-
a=x-qt-text-nam:MPEG-4 Video, streamed by the LIVE555 Media Server
a=x-qt-text-inf:para.m4e
m=video 0 RTP/AVP 96
c=IN IP4 0.0.0.0
b=AS:500
a=rtpmap:96 MP4V-ES/90000
a=fmtp:96 profile-level-id=3;config=000001B003000001B509000001000000012000C88880
07D05841214103
a=control:track1
```

Fig 3.4 Método Describe.

El cliente cuando conoce las posibilidades del servidor le envía una solicitud de DESCRIBE a la que el servidor responde con un detallado informe de todas las características del flujo multimedia.

```
Created receiver for "video/MP4V-ES" subsession (client ports 57308-57309)
Sending request: SETUP rtsp://192.168.1.33:8554/para.m4e/track1 RTSP/1.0
CSeq: 4
User-Agent: openRTSP (LIVE555 Streaming Media v2011.07.21)
Transport: RTP/AVP;unicast;client_port=57308-57309


Received 203 new bytes of response data.
Received a complete SETUP response:
RTSP/1.0 200 OK
CSeq: 4
Date: Tue, Sep 04 2012 20:49:13 GMT
Transport: RTP/AVP;unicast;destination=192.168.1.33;source=192.168.1.33;client_p
ort=57308-57309;server_port=6970-6971
Session: AA6C1740
```

Fig 3.5 Método, Setup

A continuación el cliente le envía una petición de SETUP al servidor y él responde con un PLAY.

```
Setup "video/MP4V-ES" subsession (client ports 57308-57309)
Created output file: "video-MP4V-ES-1"
Sending request: PLAY rtsp://192.168.1.33:8554/para.m4e/ RTSP/1.0
CSeq: 5
User-Agent: openRTSP (LIVE555 Streaming Media v2011.07.21)
Session: AA6C1740
Range: npt=0.000-


Received 188 new bytes of response data.
Received a complete PLAY response:
RTSP/1.0 200 OK
CSeq: 5
Date: Tue, Sep 04 2012 20:49:13 GMT
Range: npt=0.000-
Session: AA6C1740
RTP-Info: url=rtsp://192.168.1.33:8554/para.m4e/track1;seq=35138;rtptime=1427447
169


Started playing session
Receiving streamed data (signal with "kill -HUP 4782" or "kill -USR1 4782" to te
```

Fig 3.6 Método Play

Después el cliente envía un PLAY al servidor para que comience el flujo de datos. A continuación el cliente puede enviar un PAUSE para pausar la transmisión de datos o un TEARDOWN para finalizar la transmisión y liberar recursos en el servidor.

A continuación mostramos una tabla donde se pueden observar las cabeceras que contienen los diferentes métodos que se utilizan en RTSP. La tabla contiene lo que marca la IETF [8], DVB [9] y lo que utiliza LiveMedia555.

Tabla 3.2 Comparativa de importancia de las cabeceras en los métodos RTSP según IETF, DVB y en la aplicación LiveMediaServer555

RTSP Regust Header	**IETF**	**DVB requirement**	**LiveMedia Server 555**
Accept	MAY	SHOULD	OK
Accept-Language	MAY	SHOULD	KO
Bandwidth	MAY	SHOULD	KO
Content-Encoding	SHALL	SHALL	
Content-Length	SHALL	SHALL	OK
Content-Type	SHALL	SHALL	OK
Cseq	SHALL	SHALL	OK
Timestamp	MAY	N.A for LMB SHOULD for CoD	OK
If-Modified-Since	MAY	SHOULD	KO
Proxy-Required	SHALL	SHALL	KO
Range	MAY	SHOULD	OK
Require	SHALL	SHALL	KO
Scale	MAY	N.A for LMB SHOULD for CoD	OK
Session	SHALL	SHALL	OK
Transport	SHALL	SHALL	OK
User-agent	MAY	SHOULD	OK
RTSP Response Header	**IETF**	**DVB requirement**	**LiveMedia Server 555**
Alow	MAY	SHOULD	OK
Connection	SHALL	SHALL	OK
Content-Encoding	SHALL	SHALL	KO
Content-Language	SHALL	SHALL	KO
Content-Length	SHALL	SHALL	OK
Contet-Type	SHALL	SHALL	OK
Cseq	SHALL	SHALL	OK
Expires	MAY	SHOULD	OK
Last-Modified	MAY	SHALL	KO
Location	SHALL	SHOULD	OK
Public	MAY	MAY	OK
Range	MAY	MAY	OK
RTP-Info	SHALL	SHALL for RTP Streaming, N.A for UDP Streaming	OK
Scale	MAY	N.A for Lmb SHOULD MBw Tm and CoD	OK
Retry-After	MAY	SHOULD	KO
Server	MAY	SHOULD	KO
Session	SHALL	SHALL	OK
Transport	SHALL	SHALL	OK
Timestamp	MAY	SHOULD	OK
Unsupported	SHALL	SHALL	OK

3.4. Conclusiones

Después de haber analizado los métodos y las cabeceras que utiliza Live555 en comparación con lo que la IETF [8] y DVB [9] indican en sus documentos, podemos llegar a la conclusión que Live555 está diseñado a partir del estándar y de los requerimientos que la organización DVB dicta.

Se utilizan todas las peticiones necesarias exceptuando dos, de servidor a Hned, que son Announce y Redirect. Announce si que está implementado de Hned a servidor, el más común, para anunciar la descripción de una presentación identificada por una URL, en la otra dirección se utilizaría para actualizar la descripción de la sesión en tiempo real. El método Redirect lo utilizaría el servidor para redirigir al cliente a otro servidor en caso de no poder atender al cliente con los recursos necesarios. Todos los demás métodos se utilizan para realizar el diálogo correcto entre el cliente y servidor RTSP,

En el caso de las cabeceras, hay que hacer hincapié que aquí la aplicación Live555 está diseñada con los requerimientos que los programadores han visto necesarios, ya que muchas de las peticiones y respuestas no están en el programa. De las 16 cabeceras como petición que plasma el estándar se utilizan 10, y de las 21 respuestas se utilizan 15, hay un total de 6 cabeceras que Live555 no ha considerado necesario implementar para el uso del protocolo RTSP.

Se podrían dar algunos casos en los que Live555 podría ocasionar problemas. En un escenario DVB-IP compuesto por muchos clientes y por varios servidores el método Redirect sería necesario. Con esta opción un servidor colapsado podría redirigir a clientes a otros servidores menos saturados para cumplir con el servicio, con Live555 esta opción no existe. No podríamos asegurar que un único servidor pueda atender a todos los clientes ya que en algún momento de tráfico de pico podría haber clientes que no puedan acceder a los recursos del servidor.

Otra posibilidad que puede suceder es que aparezca un cliente cumpliendo con todas las cabeceras que RTSP dispone para su funcionamiento. Live555 podría ocasionar algún tipo de problemas al no entender todo lo que el cliente indica, pero podemos asegurar que muchas de ellas se pueden obviar y que Live 555 contiene las necesarias para un funcionamiento correcto.

Llegamos a la conclusión de que Live555 es una aplicación diseñada a partir del estándar y del consorcio DVB, ya que en general cumple con todos los requisitos. A nivel de peticiones Live555 utiliza todas las que el estándar y el consorcio DVB creen necesarias, exceptuando Announce y Redirect, de servidor a HNED, y a nivel de cabeceras utiliza un número inferior a las que ambas organizaciones creen oportunas. Esto no impide la compatibilidad con otras aplicaciones que no forman parte del conjunto de programas de prueba de Live555, como VLC[24] o Kaffeine [43]. Con ambas la reproducción de contenidos multimedia bajo demanda ha funcionado perfectamente utilizando RTSP como protocolo de diálogo entre ambos.

Después de las diferentes pruebas creemos que Live555 es una aplicación óptima y de gran funcionalidad en un escenario DVB-IP, ya que podemos utilizarla para servir a diferentes clientes de video bajo demanda, cumpliendo con los requisitos que el estándar y el consorcio DVB piensan para un escenario óptimo.

CAPÍTULO 4. Protección de flujos en DVB-IP

4.1. Introducción

En el estándar DVB-IP la protección de datos es una parte básica del funcionamiento del sistema. DVB-IP utiliza UDP como protocolo de transporte, lo que implica que para ofrecer fiabilidad (dado que la transferencia de datagramas no es fiable, al no efectuarse retransmisiones en la capa de transporte) se deben establecer mecanismos en la capa de aplicación. Esto conlleva el diseño de aplicaciones que sean capaces de resolver las posibles pérdidas de paquetes utilizando diferentes métodos.

En DVB-IP se plantean dos posibles soluciones para evitar las pérdidas de paquetes multimedia: retransmisión de paquetes RTP [11] y AL-FEC [2] (Application Layer Forward Error Correction). En nuestro proyecto hemos diseñado un servidor de retransmisión capaz de atender peticiones de retransmisión RTP por parte de los clientes y un servidor AL-FEC (siguiendo las recomendaciones de SMPTE [47]) capaz de enviar un flujo de paquetes FEC al cliente para que éste pueda recuperar los paquetes en caso de pérdida. Ambos módulos se han diseñado como subsistemas independientes del servidor de streaming principal, para no sobrecargarlo y permitir su instalación en otras máquinas, además de aportar modularidad al diseño.

Ambas posibilidades, retransmisiones y AL-FEC, son contempladas para evitar las pérdidas de paquetes. Cada una tiene ventajas e inconvenientes:

a) Un servidor de retransmisión es eficiente si se evalúa la eficiencia de uso canal ya que el aumento del consumo del ancho de banda solo se da en caso de pérdidas (en el momento de enviar retransmisiones), mientras que cuando no hay pérdidas se utiliza solo el ancho de banda necesario para el streaming.

   Por el contrario, en el caso de que la red sea muy extensa, los retardos de retransmisión serán elevados, lo que puede llevar a que el RTT (round trip time[1]) sea excesivo y las retransmisiones sean inútiles, ya que llegarán demasiado tarde como para ser usadas en la reproducción (el terminal receptor ya habrá tenido que presentar la imagen dañada o el fragmento de audio perdido).

b) En cambio AL-FEC se basa en utilizar la redundancia de los datos, enviando un flujo secundario al cliente con el cual éste sería capaz de procesar y recuperar paquetes perdidos. AL-FEC utiliza el mismo canal utilizando otro puerto, lo que lleva a ocupar permanentemente un ancho de banda adicional (aunque más pequeño que el flujo principal) y también se incrementa el procesamiento de datos en el cliente aunque

[1] Round Trip time es el tiempo que tarda un paquete des de que sale del emisor hasta que vuelve a éste pasando por el receptor.

no haya pérdidas, ya que éste tiene que estar recibiendo constantemente los datos redundantes.

Por otro lado este sistema está limitado a solucionar un número de pérdidas, en función de lo que el programador estime y la complejidad que crea oportuna, con el retraso de procesamiento que conlleva.

4.2. Retransmisiones RTP

Las retransmisiones rápidas son un mecanismo de protección en el que el cliente, al detectar pérdidas de paquetes, envía un paquete tipo RTCP Feedback (indicado por el estándar de la IETF [14] y la organización DVB [2][11]) donde indica el paquete que ha perdido. El servidor, al recibir la petición de cualquier cliente, analiza el paquete y comprueba si él dispone de dicho paquete para reenviar al grupo multicast de retransmisiones o vía unicast (ambas posibilidades son aceptadas, aunque la opción multicast es más escalable).

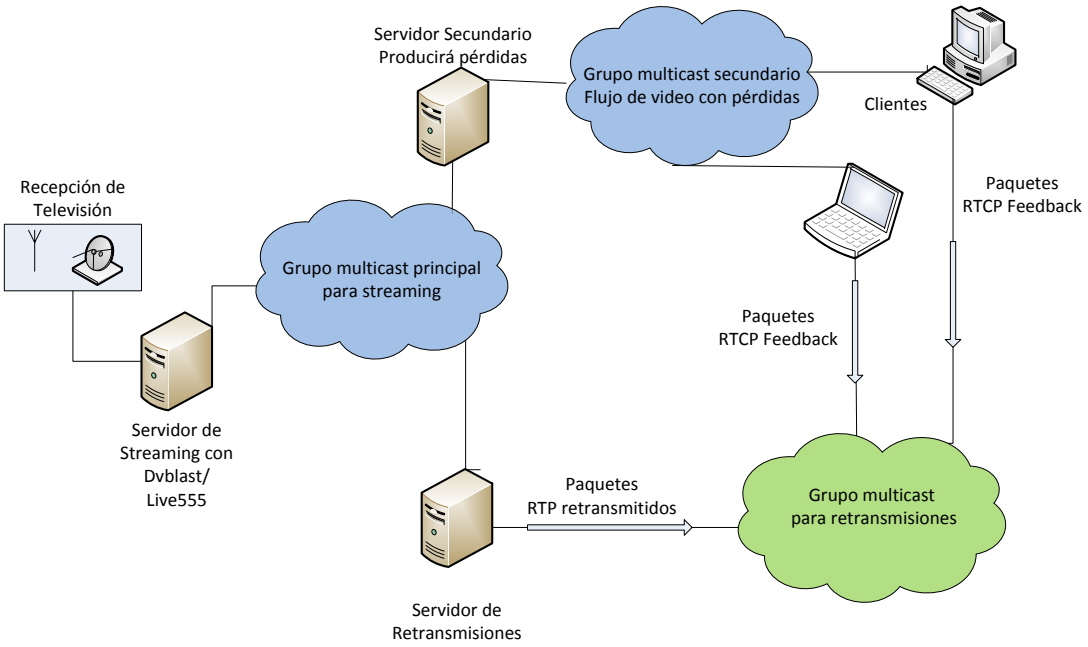

Fig 4.1 Escenario de retransmisiones RTP

En la figura anterior se muestra el escenario que hemos utilizado para recrear el funcionamiento de las retransmisiones RTP. El escenario incluye un servidor principal que emitirá flujos de video, el servidor de retransmisiones que atenderá las peticiones de los clientes y el servidor secundario que será el encargado de provocar pérdidas en el flujo de video.

4.2.1. Paquete RTCP Feedback

Para las retransmisiones el estándar establece la utilización de un paquete RTCP específico, el RTCP Feedback. En la figura 4.2 mostramos la cabecera de este paquete y comentaremos la función de cada campo.

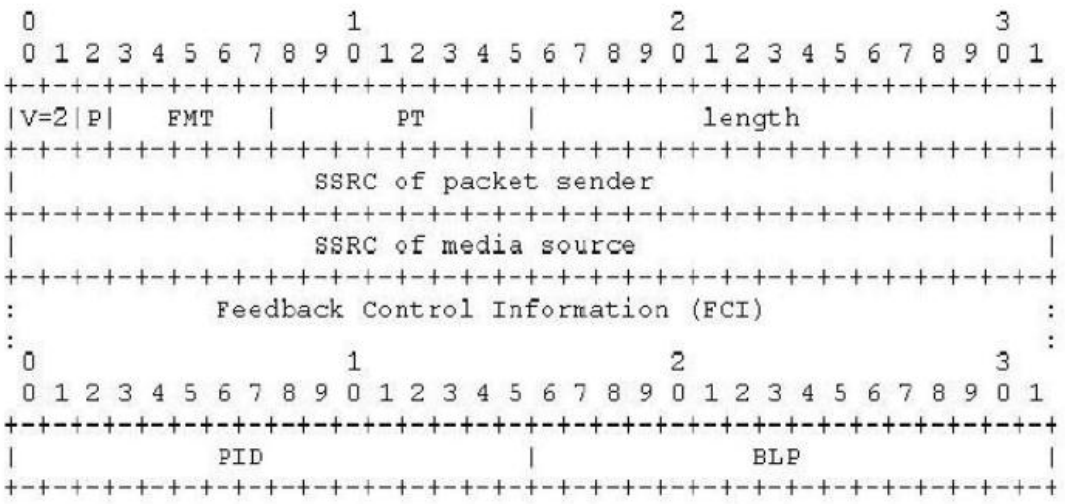

Fig 4.2 Paquete para peticiones de retransmisión RTCP Feedback [2].

Los campos del paquete Feedback de la figura 4.2 son los siguientes:

- V (2 bit): Indica la versión de RTP, actualmente es dos.

- P (1 bit): Padding, se establece a 1 si el paquete contiene información de relleno. Se utilizará cuando el paquete no llegue al tamaño mínimo para poder ser transportado. El receptor desechará los octetos sobrantes.

- FMT (5 bits): Indica el tipo de mensaje que contiene el paquete Feedback:

  0: No asignado

  1: Picture Loss Indication (PLI). Con este mensaje de pérdida de imagen, un cliente informa al servidor de la pérdida de una cantidad indefinida de datos de vídeo codificados que pertenecen a una o más imágenes.

  2: Slice Loss Indication (SLI). Con este mensaje un cliente puede informar a un servidor que se ha detectado la pérdida o corrupción de uno o varios macrobloques[2] consecutivos.

[2] Los macrobloques son la unidad básica sobre la cual se realiza la compensación de movimiento. El objetivo de las técnicas de codificación de este tipo es la reducción de la tasa de bits, analizando la redundancia y codificando la mínima información necesaria

3: Reference Picture Selection Indication (RPSI). Con este mensaje se indica que hay una imagen de referencia conocida. Este método es utilizado por el codificador cuando éste detecta una pérdida de sincronía, entre él y el decodificador. En la información útil se indica el ID temporal de la imagen dañada y el tamaño de la región dañada. Éste mecanismo es utilizado por las versiones mas modernas de codificación, tal como MPEG-4 y H.263 v2.

4-14: No asignado

15: Application layer FB (AFB) message. Son un caso especial de carga útil específica. Estos mensajes se utilizan para el transporte de datos definidos por la aplicación directamente. Los datos que se transportan no están identificados por el mensaje de FB. Por lo tanto, la aplicación debe ser capaz de identificar la carga del mensaje.

16-30: No asignado

31: Reservado para una futura implementación.

- PT (8 bits): Identifica éste paquete como RTCP Feedback. La IANA[48] especifica dos valores:
    - Valor 205, paquete RTPFB (transport layer feedback message), este valor se utiliza para la petición de retransmisiones.
    - Valor 206, paquete PSFB, payload specific feedback message
- Length (16 bits): Indica la longitud del paquete incluyendo la cabecera pero no el relleno.
- SSRC of packet sender (32 bits): Identificador de sincronización de la fuente del paquete RTCP Feedback, la fuente que origina éste paquete.
- SSRC of media source (32 bits): identificador de la fuente emisora del paquete original que se ha perdido.
- Feedback Control Information (FCI): Información adicional para el mensaje feedback.
- PID (16 bits): indica el número de secuencia del paquete perdido.
- BLP (16 bits): Se utiliza para indicar que se han perdido los paquetes siguientes al indicado en el PID. Si el bit i de la máscara se pone a 1, el remitente no ha recibido el paquete RTP (PID+i)(módulo 2^16).

Ejemplo de utilización de la cabecera BLP: si se ha perdido el paquete con número de secuencia 25633 y el 25634, el BLP será 0000000000000001, lo que nos indicará que hemos perdido el paquete:

$$25633 + 000000000000001(\text{modulo } 2^{16}) = 25634 \qquad \textbf{(4.1)}$$

En este proyecto hemos implementado un servidor de retransmisiones y un cliente que se adaptan al estándar de la IETF [14] y la organización DVB [2][11]. Hemos utilizado los paquetes RTCP feedback para pedir retransmisiones al servidor que hemos diseñado. No hemos implementado la funcionalidad del campo BLP de la cabecera del paquete FEC, sino que por cada paquete perdido se envía una petición de retransmisión. Por este motivo hemos diseñado una escucha previa del canal de retransmisiones para no inundar la red con mensajes repetidos. Esta parte la describiremos mejor en la parte de diseño del cliente de retransmisión.

4.3. AL-FEC

Es un tipo de mecanismo de corrección de errores que permite su corrección en el receptor sin tener que realizar retransmisiones de la información original. Se utiliza en sistemas sin retorno o sistemas en tiempo real donde no se puede esperar a la retransmisión para mostrar los datos.

En este proyecto hemos implementado AL-FEC (Application Layer-Forward Error Correction) como técnica de corrección. Esta técnica ofrece una buena protección contra las pérdidas de paquetes a un coste de complejidad razonable. Consiste en un código de paridad calculado a partir de una matriz bidimensional de entrelazado de paquetes RTP. Los paquetes de corrección se envían en un flujo distinto al flujo principal de datos.

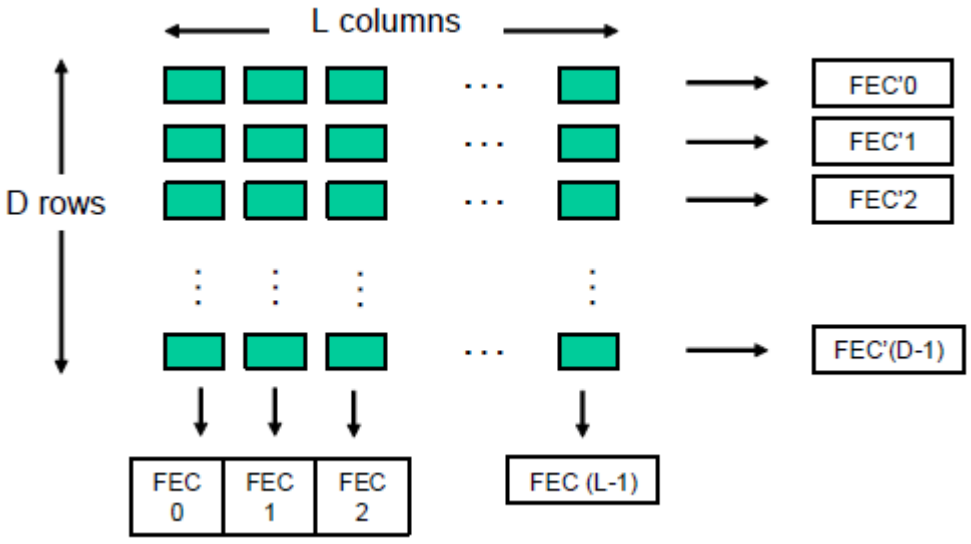

Fig 4.3. AL-FEC 2D implementado en éste proyecto [13]

El método permite la corrección de hasta un error por fila o columna, lo que posibilita recuperar situaciones “complicadas” como la ilustrada en la figura 4.4, pero en cambio fallaría si perdiéramos simultáneamente los paquetes 4, 5, 10 y 11, puesto que no hay manera de reparar ninguna fila o columna con dos fallos, de manera aislada.

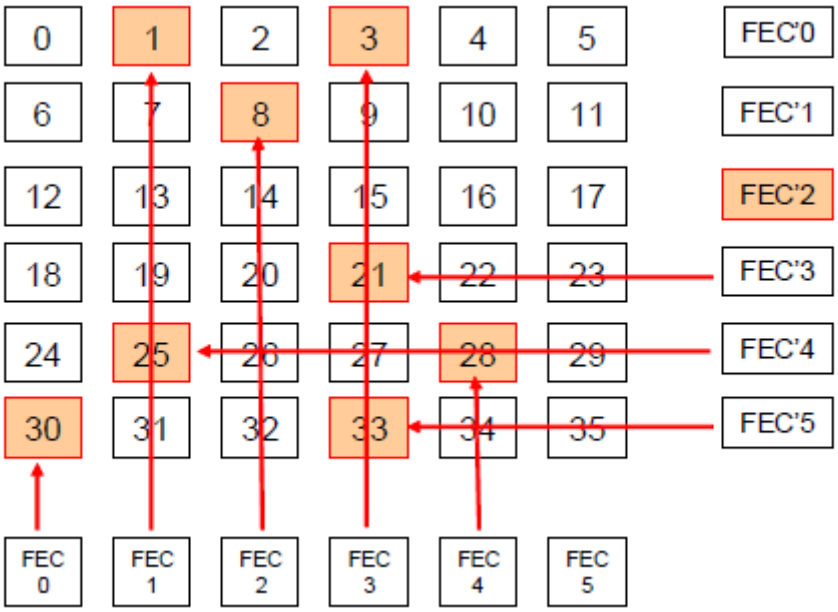

Fig 4.4 Ejemplo de situación recuperable en caso de pérdidas. Los paquetes sombreados son las pérdidas [13]

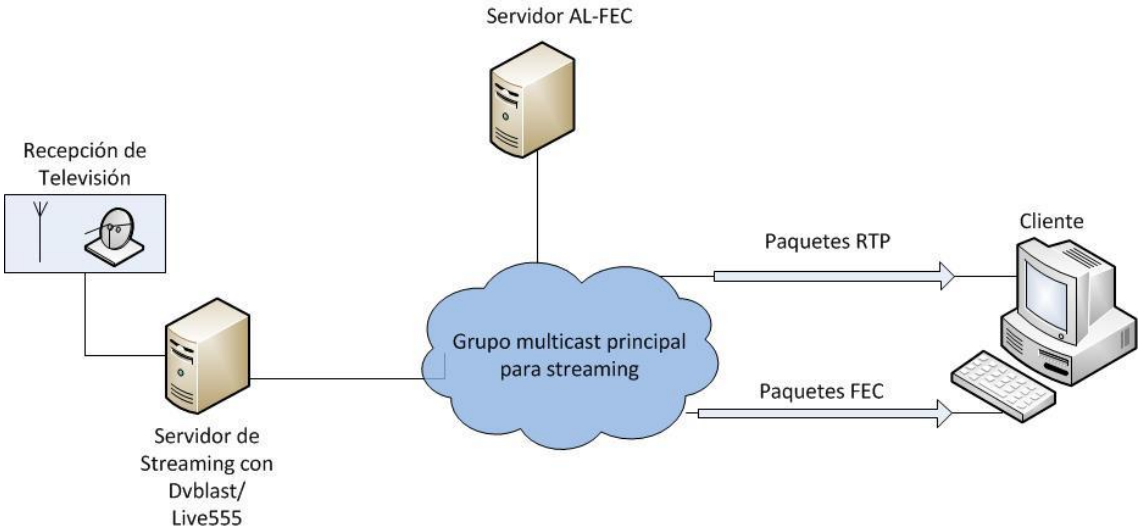

Fig 4.5 Escenario con servidor AL-FEC

En la figura 4.5 se muestra el escenario que hemos diseñado para realizar pruebas con nuestro servidor de AL-FEC.

4.3.1. Paquete FEC

Los campos de la cabecera del paquete FEC son los siguientes:

- SN base: número de secuencia del primer paquete protegido.
- Length Recovery: Se utiliza para determinar la longitud de cualquier paquete recuperado.
- E: indica si hay extensión de cabecera, si se establece a 0 no habrá extensión.
- Payload Type Recovery: se obtiene a partir de la operación de protección de los campos de tipo de carga de los paquetes RTP.
- Mask: Indica los paquetes protegidos en el paquete FEC.
- Time Stamp Recovery: se obtiene a partir de la operación protección aplicada al campo Timestamp.

La figura 4.5 muestra los diferentes campos explicados en el apartado anterior que contiene la cabecera de un paquete FEC.

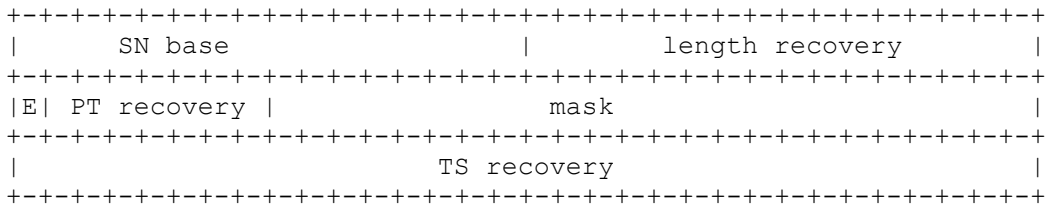

Fig 4.5 Cabecera de un paquete FEC [2]

La figura 4.6 muestra el conjunto de cabeceras que se envían cuando enviamos un flujo de protección de datos utilizando AL-FEC. Podemos observar que el paquete está formado por la cabecera RTP, la cabecera FEC y los datos FEC, donde se incluyen la operación binaria XOR de los paquetes protegidos.

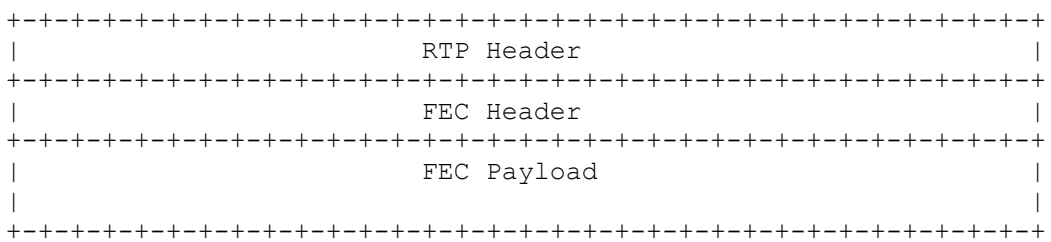

Fig 4.6 Formato del paquete FEC RTP [2]

4.3.2. Operación de protección

A continuación definiremos como se protegen los paquetes RTP utilizando la operación binaria XOR. La operación XOR se realiza a partir de los campos de la cabecera RTP y de los datos que contiene el paquete.

Length Recovery: en este campo sumaremos a partir de la operación XOR la longitud de todos los paquetes RTP que vamos a proteger. Para realizar el cálculo necesitamos la longitud de los datos útiles, la lista CSRC, extensión y el relleno si tuviese. A partir de aquí no importa que haya paquetes de diferentes tamaños, ya que se podrá recuperar el tamaño de la carga útil a partir del procedimiento de recuperación en caso de pérdidas.

Payload Type Recovery se formará cogiendo los valores de carga útil de la cabecera de los paquetes RTP y aplicando la operación XOR a dichos valores. A partir de aquí se podrá recuperar el tipo de formato del paquete perdido.

El campo de la mascara no codificará ningún valor, pero indicará que paquetes se han protegido a partir del SN base que indicará el primer paquete protegido. En la mascara se pondrá a 1 los bits para indicar los paquetes protegidos, siendo SN+i los paquetes protegidos.

En el campo Time Stamp Recovery se codificarán todas las marcas de tiempo de los paquetes RTP protegidos, realizando la operación XOR con los 32 bits que forman esta marca, así se podrá recuperar la marca de tiempo del paquete que hayamos perdido.

La carga útil del paquete FEC se consigue haciendo la operación XOR a la lista CSRC, extensión RTP, la carga útil del paquete y el relleno si tuviese. Como podemos ver son los mismo parámetros de la recuperación de longitud, de ésta forma podremos comparar el valor y corroborar que los datos que hemos obtenido son correctos.

CAPÍTULO 5. Desarrollo del software para retransmisiones RTP y AL-FEC

5.1. Escenario

Nuestro escenario se basa en un servidor de streaming utilizando la aplicación dvblast para el perfil LMB (Live Media Broadcast) y Live555 para el CoD (Content on Demand). Los dos módulos diseñados se encuentran instalados y ejecutándose en el servidor principal (servidor de streaming), pero se han diseñado como partes independientes, de ahí que en la figura siguiente aparezcan como módulos separados.

En el escenario siguiente, el grupo multicast principal (aquél en el que se transmite el contenido a proteger) es el 224.0.1.2 con puerto 5004 y el grupo que transmite las retransmisiones es el 224.100.100.100:15004. El flujo de paquetes AL-FEC se transmite por el grupo principal aumentando en dos unidades el puerto destino. El escenario mostrado en la figura 5.1 incluye los siguientes elementos: dos antenas terrestres, dos antenas satélite, un servidor de streaming, el servidor AL-FEC, el servidor de retransmisiones y dos clientes, un PC portátil personal y un PC del laboratorio 325 de la EETAC.

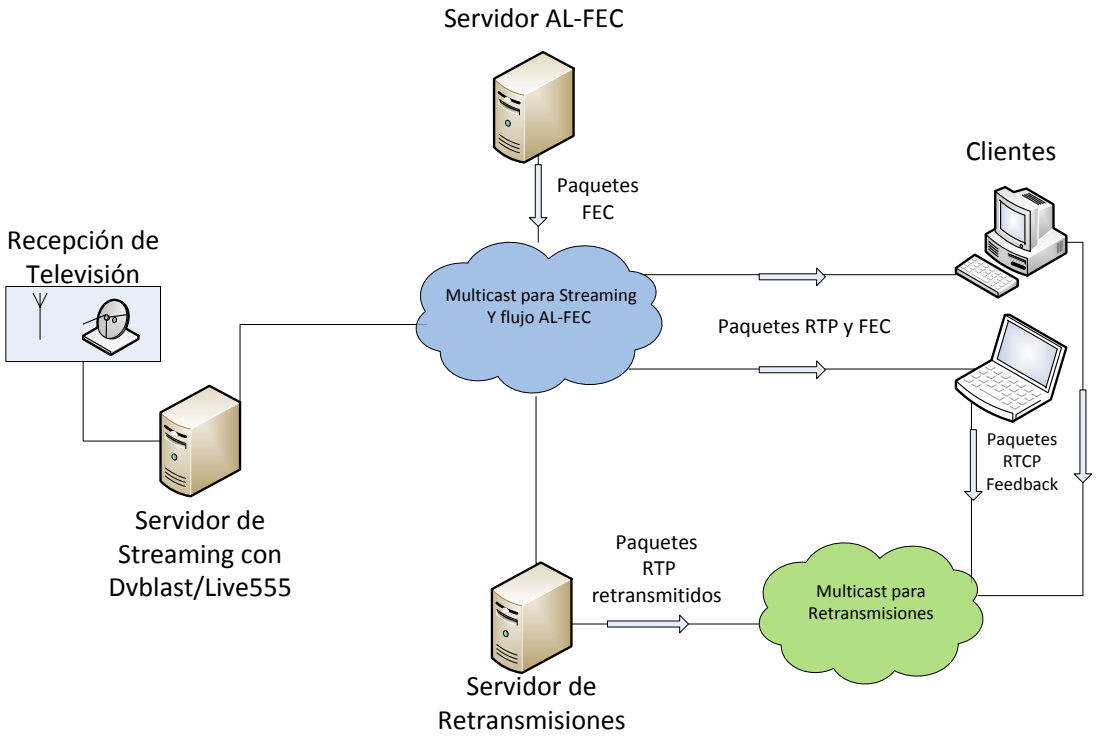

Fig 5.1 Escenario de pruebas de AL-FEC y retransmisiones.

5.1.1. Dvblast

Dvblast [27] es una aplicación diseñada por VLC para emitir streaming a partir de una aplicación ligera y eficiente. Es capaz de retransmitir flujos recibidos por tarjetas DVB-S y DVB-S2 (Satélite), DVB-T (Terrestre) y transmitir sobre UDP y RTP. Es una aplicación de código abierta donde podemos trabajar con el código y modificarlo para alcanzar nuestros objetivos.

Nosotros hemos utilizado este software para realizar el perfil de LMB y emitir contenidos capturados por nuestra tarjeta sintonizadora DVB-T a través de un grupo multicast por la red de la escuela. Nuestro servidor situado en el sótano de la EETAC (instalaciones de I2CAT) dispone de cableado de antenas terrestres y satélites, y desde nuestro laboratorio nos conectamos por SSH y emitimos los flujos de video capturados por las tarjetas sintonizadoras.

La secuencia para iniciar la aplicación es la siguiente:

dvblast -a 0 -c /home/dani/Escritorio/configuracion2.conf -f 794000000 -m qam_64 -b 8

-a = El adaptador instalado en el servidor.
-c = El archivo de configuración que más abajo comentaremos.
-f = la frecuencia del canal terrestre que queremos emitir.
-m = la modulación, en este caso QAM 64.
-b = El ancho de banda en MHz, esta vez 8Mhz

El archivo de configuración debe contener los siguientes parámetros.

239.100.0.1:50001 1 801

239.100.0.2:50001 1 802

239.100.0.3:50001 1 803

Una dirección con un puerto para emitir lo capturado por la tarjeta sintonizadora y el PID del canal que queremos emitir, en este ejemplo son los canales emitidos por TV de Cataluña (TV3, C33, Esports3…) en el multiplex de TDT situado en el canal UHF de 794 Mhz.

5.1.2. Live555

Live555 [25] es un conjunto de aplicaciones diseñadas para streaming de video y audio. También contiene un conjunto de herramientas para testear algunos protocolos como pueden ser RTSP o SIP. Nosotros hemos utilizado un servidor de video diseñado por Live555 para realizar el video bajo demanda.

La interfaz de la aplicación es sencilla, una vez guardados los videos en la raíz del servidor se puede acceder a ellos des de cualquier ordenador utilizando la URL RTSP. En la siguiente imagen mostramos la interfaz del servidor de video Live555.

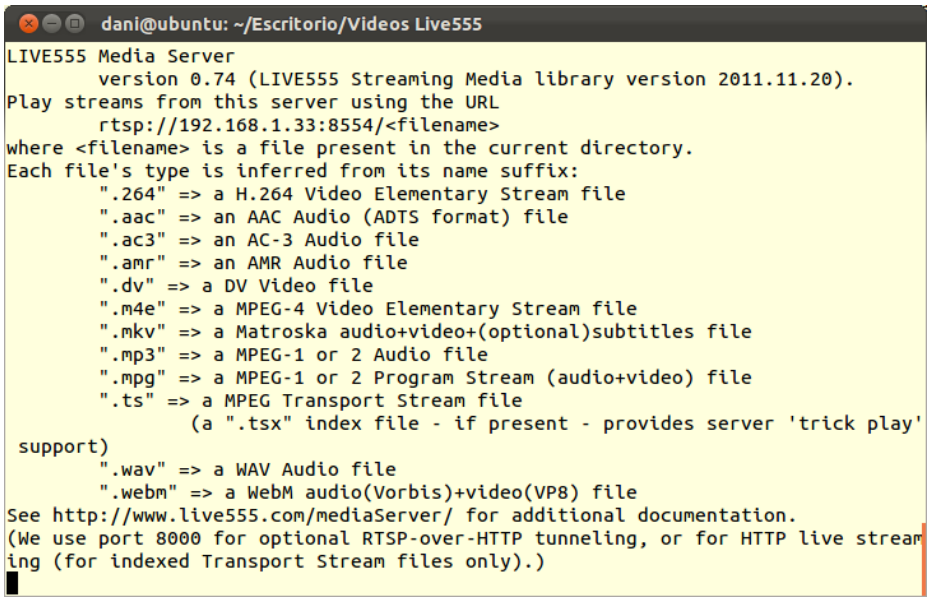

Fig 5.2 Live555 Media Server iniciado

El objetivo de nuestras pruebas era gestionar video bajo demanda, y las pruebas consistieron en copiar videos en la raíz del servidor y luego con un cliente VideoLAN [24] abrir la URL rtsp://192.168.1.33:8554/NombreDelArchivo y abrir el video que queríamos visualizar.

5.1.3. VLMa

VLMa [28] es una aplicación para gestionar las transmisiones de canales de televisión, digitales recibidos a través de vías terrestres o por satélite. Su interfaz se ofrece como un sitio web escrito en Java. También es capaz de reproducir archivos de audio y vídeo.VLMa consiste en un demonio (llamado VLMad) y una interfaz web (llamado VLMaw). Con ésta aplicación podemos gestionar los servicios de un servidor de DVB a partir de una aplicación web.

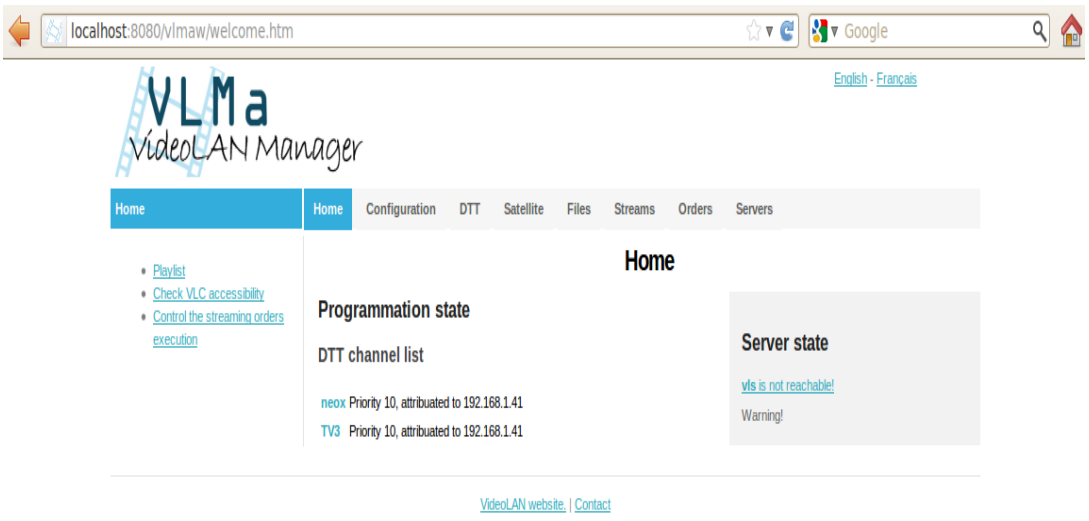

Fig 5.3 Interfaz principal de VLMA.

5.2. Servidor y cliente de retransmisión RTP

El servidor que hemos diseñado para la prevención de pérdidas de paquetes de video se basa en la norma establecida por DVB-IP de retransmisiones rápidas [2].

El servidor es un módulo independiente del servidor de streaming, lo que nos da la posibilidad de instalarlo en cualquier punto de nuestra red. El servidor se mantiene a la escucha del grupo multicast principal 224.0.1.2:5004 y del grupo multicast de retransmisiones 224.100.100.100:15004 para atender las peticiones de clientes con pérdidas. Cuando recibe una petición la procesa y reenvía el paquete RTP que le han pedido.

El cliente estará escuchando de la multicast principal y comprobando que no hay pérdidas de paquetes, ya que en caso de pérdidas enviará peticiones de retransmisión al servidor utilizando el paquete RTCP Feedback.

Ambas aplicaciones las hemos diseñado a partir de una aplicación libre llamada dumprtp [49] cuyo objetivo es conectarse a un grupo multicas y recibir flujos RTP.

5.2.1. Diseño del servidor

A la hora de diseñar nuestro servidor de retransmisiones nos hemos basado en lo que el estándar de la IETF [14] y la organización DVB [2][11] marcan para el uso de retransmisiones en DVB-IP.

La aplicación se ha diseñado a partir de threads; así somos capaces de realizar dos acciones diferenciadas y conseguimos un rendimiento eficaz para el servidor que debe escuchar por dos sockets a la vez.

a) El primer thread actúa como un cliente cualquiera, escuchando en la dirección multicast principal y almacenando los paquetes recibidos en un buffer del tipo char de 1000x1320 posiciones, para guardar cada paquete recibido en cada fila. Ya que cada paquete IP contendrá 7 paquetes Transport Stream de 188 bytes, lo que nos da 1316, mas la cabecera RTP de 4 bytes son un total de 1320 bytes útiles para almacenar.

b) El segundo thread lo mantenemos a la escucha en la dirección multicast de peticiones de retransmisión.

En el caso de recibir una petición de retransmisión, actúa de la siguiente manera:

a) se bloquea el acceso a la variable de almacenamiento “buffer”, declarada previamente como una variable global, para poder bloquear y desbloquear el acceso a ésta en función del thread que la necesite.

b) analizamos el número de secuencia que nos pide, lo buscamos en el buffer de almacenamiento.

c) si encontramos el paquete que nos pide lo enviamos a la dirección multicast para retransmisiones, si no lo tenemos enviamos un paquete con el campo de número de secuencia en NULL.

d) Una vez terminada la petición desbloqueamos el acceso a la variable global y continuamos escuchando a la dirección multicast de envío de peticiones de retransmisiones, y el thread principal continúa almacenando los datos recibidos.

5.2.2. Diseño del cliente

El cliente de retransmisiones se basa en un cliente que escucha a partir de un socket los paquetes de datos de una dirección multicast principal y va analizando los números de secuencia para comprobar que siguen un orden secuencial y que no falta ninguno. Si el orden es correcto los va almacenando en un buffer. En el caso de que falten paquetes el cliente llama un thread para que abra un socket secundario encargado enviar la petición de retransmisión mediante un paquete específico del tipo RCTP feedback. Al thread le pasaremos en una variable char el número de secuencia del primer paquete perdido, la posición en la variable buffer y el número total de paquetes perdidos. Al entrar en la función del thread se analizará la variable y obtendremos los diferentes valores para saber los paquetes perdidos y la posición que ocupan en el buffer de almacenamiento.

Antes de enviar la petición de retransmisión el cliente escuchará el socket de envío de peticiones de retransmisiones durante un tiempo aleatorio de 0 a 2 segundos. Este valor los hemos fijado nosotros, ya que en el estándar [2] indica que el valor de este tiempo de espera deber ser fijado en el SD&S. El proveedor de servicios utilizará el que crea conveniente, para evitar inundar la red de peticiones de retransmisiones repetidas. En el caso de no escuchar ninguna petición enviará la suya. Si escucha una petición de retransmisión, la analizará para comprobar que no es una de las suyas. Si por lo contrario es la misma petición que va a enviar, borrará la suya y se pondrá a escuchar el socket para recibir el paquete (que otro cliente ha solicitado antes que él), y después volverá a comenzar hasta que haya enviado todas las peticiones que necesitaba.

Otra posibilidad es que escuche un paquete retransmitido. En este caso analizará si este paquete iba a ser una petición de las suyas, y si es así, lo guardará en el buffer y borrará la petición de retransmisión.

Cuando reciba la respuesta a su petición con un paquete de RTP retransmitido, bloqueará el acceso a la variable buffer global para guardar el paquete en su posición, y poder seguir el orden lógico de los números de secuencia. Si el servidor no dispone del paquete envía un NULL en el campo de secuencia.

5.2.3. Pruebas realizadas

Para realizar las pruebas hemos diseñado una aplicación que realiza la función de servidor secundario, capaz de recibir los paquetes y reenviar sólo una parte de ellos, simulando así pérdidas producidas por a red. El cliente escucha en la dirección que retransmite este servidor secundario, y cada vez que observa pérdidas de paquetes envía peticiones de retransmisión al servidor de retransmisiones.

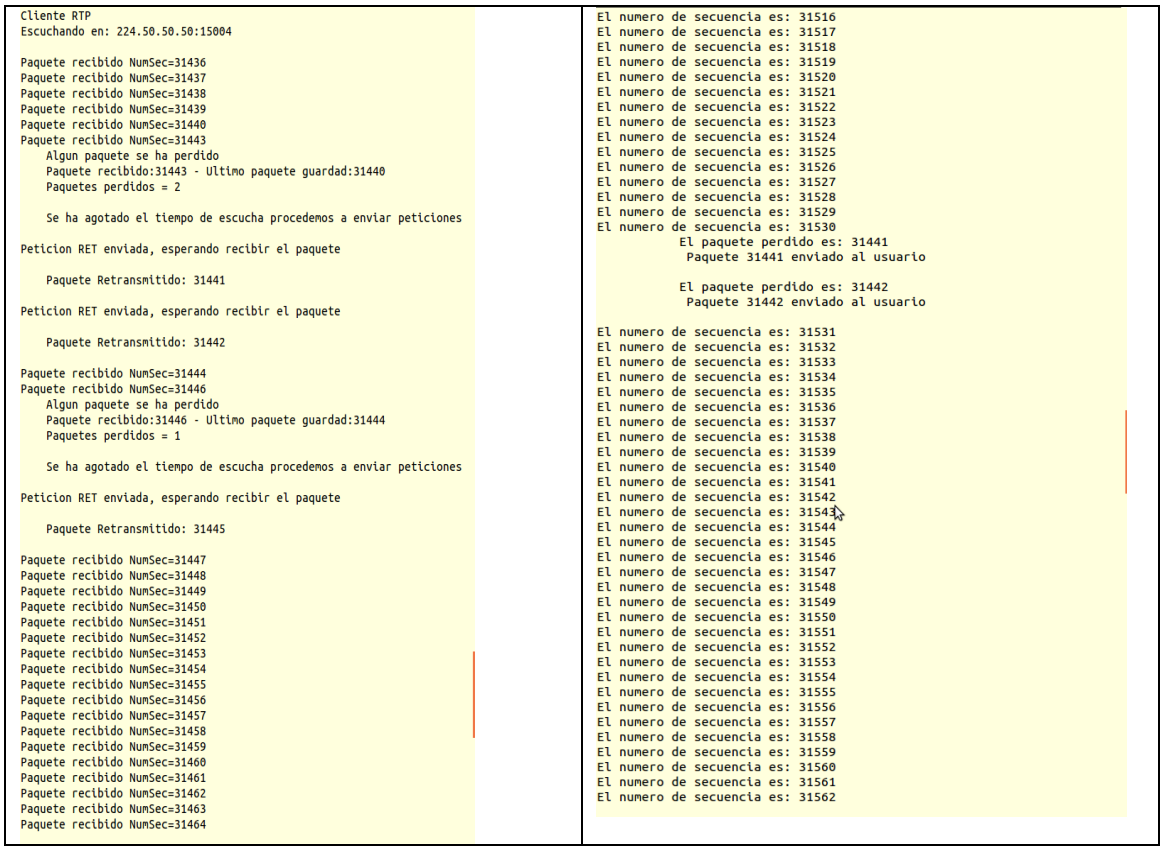

Fig 5.5 Captura del cliente RTP y del servidor RTP, realizando retransmisiones

En la figura 5.5 se muestra como las dos aplicaciones cliente/servidor interactúan en caso de pérdidas detectadas por el cliente. El cliente detecta pérdidas, se espera un tiempo aleatorio entre 0-2 segundos y envía la petición RTCP feedback al servidor que le responde enviándole el paquete solicitado. Tanto el cliente como el servidor dejan de atender al thread principal cuando tienen que atender al canal de retransmisiones.

Cuando el cliente recibe el paquete retransmitido lo guarda en la posición correspondiente en el buffer de almacenamiento, ya que previamente ha guardado la posición del paquete perdido. De esta forma no perdemos el orden de los números de secuencia y el flujo de video se podrá reproducir con normalidad.

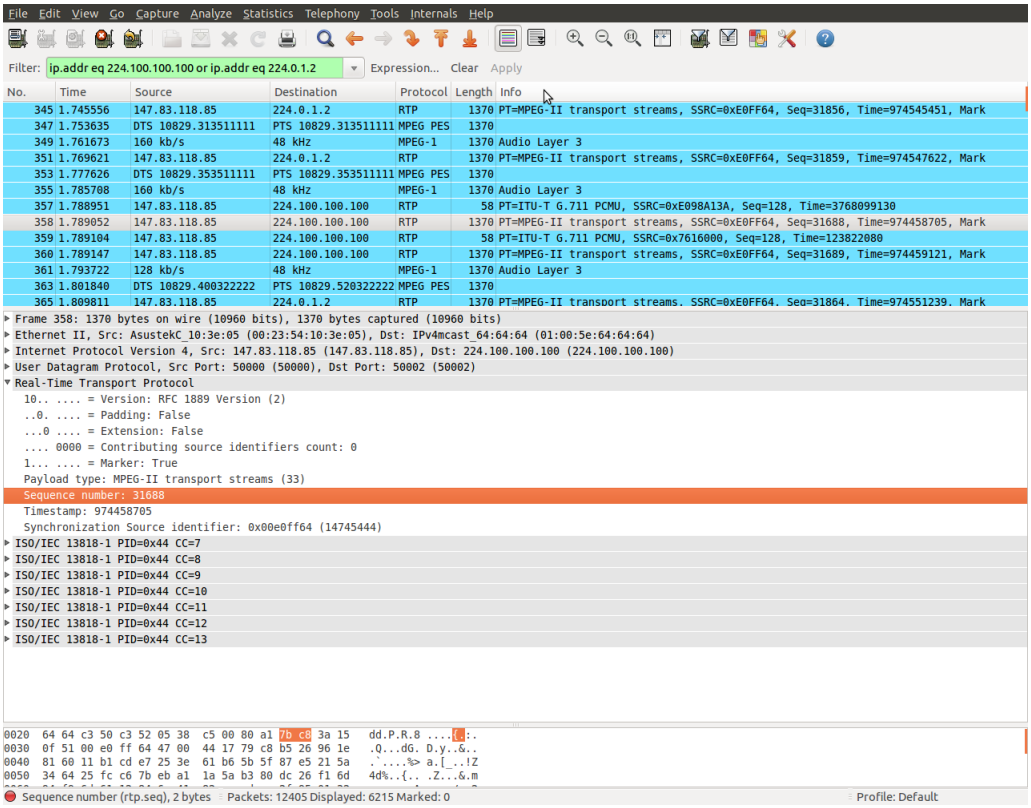

Fig 5.6 Captura de wireshark, 2 peticiones de retransmisión y los 2 paquetes retransmitidos

En la figura 5.6 podemos observar como entre el flujo de paquetes RTP a al grupo multicast 224.0.1.2 (grupo multicast para streaming) hay 4 paquetes que se dirigen al grupo multicast 224.100.100.100 de retransmisiones. Si observamos el tamaño de cada paquete podemos diferenciar los 58 bytes del paquete feedback y los 1370 del paquete RTP retransmitido.

En la figura 5.7 hemos comprobado que la aplicación puede trabajar con varios clientes que sufren las mismas pérdidas. Hemos ejecutado dos clientes simultáneos que reciben las mismas pérdidas en el mismo momento. Uno de los clientes, al escuchar el medio antes de enviar la petición de retransmisión, escucha otra petición indicando la misma pérdida que él ha tenido. Para no enviar peticiones duplicadas, el cliente no envía su petición sino que se dispone a escuchar el medio otra vez de manera indefinida para recibir el paquete retransmitido que otro cliente ha pedido por él. Esto es debido al tiempo aleatorio entre 0 – 2 segundos que los clientes escuchan el medio antes de enviar su petición de retransmisión, es un mecanismo de seguridad para no inundar la red con peticiones repetidas.

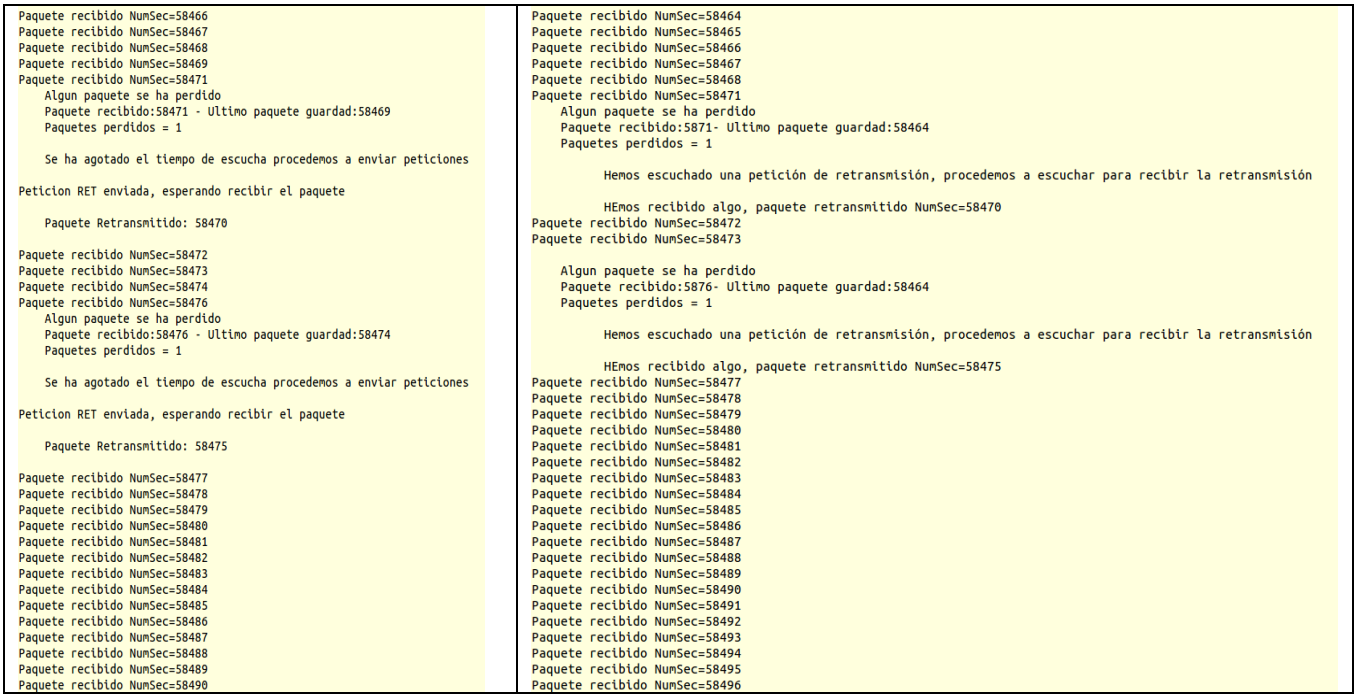

Fig 5.7 Dos clientes con las mismas pérdidas

5.3. Servidor FEC

Las operaciones para protección del flujo de datos audiovisual se realizan en el servidor que hemos modificado de otro estudiante [1], que realizó una versión inicial de AL-FEC limitada a una dimensión (es decir, un paquete de corrección por cada N paquetes originales, mediante la operación XOR bit a bit sobre los paquetes RTP).

Los pasos a seguir por el servidor FEC son los siguientes:

1. Recibimos los paquetes RTP de la fuente original y los introducimos en una matriz bidimensional, donde solo almacenamos la cabecera RTP y el payload.

2. Realizamos la operación XOR con los paquetes de cada fila y los paquetes de cada columna, dando como resultado un nuevo paquete.

   La operación se realiza de la siguiente forma:

   El primer campo que formaremos de la cabecera FEC será el length recovery, a partir de la suma de la longitud en bits del campo de la cabecera RTP CSRC, extensión, relleno y los datos útiles.

   En el campo de la cabecera payload type recovery, sumaremos los bits de la parte de la cabecera de los paquetes RTP que indican el tipo de formato, concretamente el campo payload type.

En el campo de máscara y el SN base se rellenará al principio. En el SN indicaremos el primer paquete protegido y en la máscara los demás paquetes protegidos en función del número que el usuario ha introducido al iniciar la aplicación.

En el campo Timestamp Recovery se codificarán todas las marcas de tiempo de los paquetes RTP protegidos, realizando la operación XOR con los 32 bits que forman esta marca. Así se podrá recuperar la marca de tiempo del paquete que hayamos perdido.

La última parte será la carga útil del paquete FEC, que conseguiremos haciendo la operación XOR a la lista CSRC, extensión RTP, la carga útil del paquete y el relleno si tuviese. Como podemos ver son los mismo parámetros de la recuperación de longitud, de ésta forma podremos comparar el valor y corroborar que los datos que hemos obtenido son correctos.

3. Empaquetamos los datos resultantes de la operación con una cabecera RTP y los enviamos al cliente, al mismo grupo multicast que los datos originales per aumentando el numero de puerto en dos unidades.

A la hora de enviar los paquetes FEC hay diversas formas de hacerlo:

- **Envío constante y no entrelazado:** En éste modo los paquetes se envían en un flujo a parte de datos de manera constante, siempre se envían los paquetes de datos antes que los paquetes de reparación.

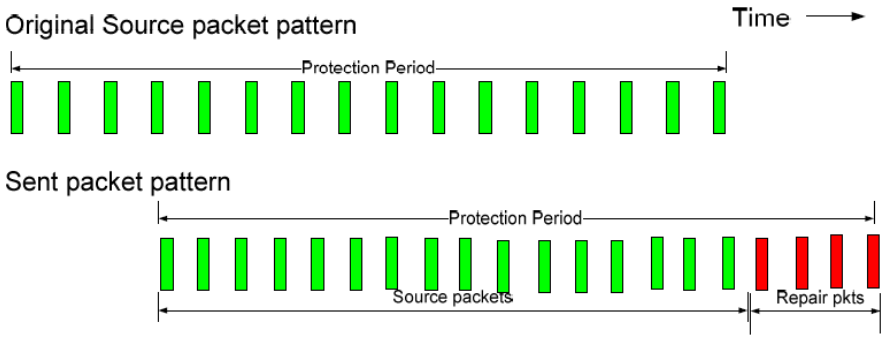

Fig 5.8 Envío no entrelazado [12].

- **Envío completamente entrelazado**: Se mantiene una tasa de envío más o menos constante, y los paquetes de recuperación se envían cada cierto bloque de paquetes originales.

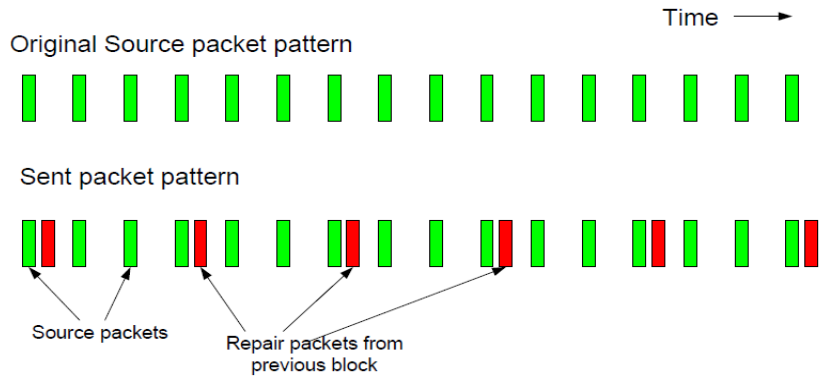

Fig 5.9 Envío completamente entrelazado [12]

- **Envío parcialmente intercalado*:*** En éste caso los paquetes se envían intercalados con el flujo original.

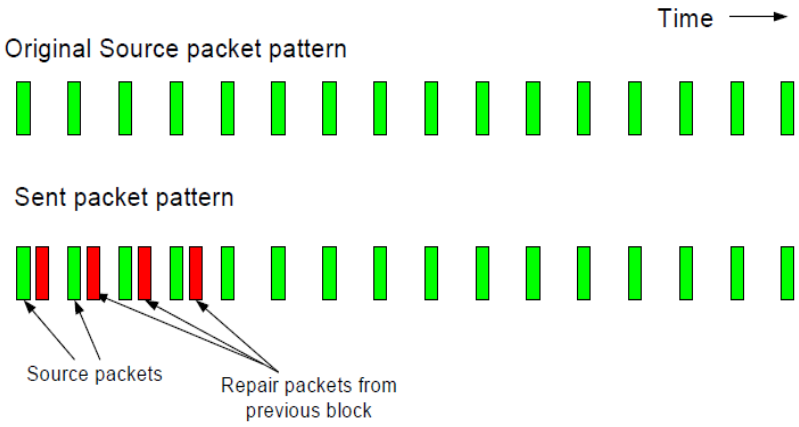

Fig 5.10 Envío parcialmente intercalado [12]

En nuestro caso nuestra aplicación realiza el envío a velocidad constante y no entrelazado, es decir, el que muestra la figura 5.8, los paquetes FEC son enviados después del flujo de paquetes multimedia. La implementación consiste en enviar un flujo de datos FEC secundario al flujo principal de datos. Todo va dirigido al mismo grupo multicast pero en un puerto situado dos unidades por encima, tal como indica el estándar DVB-IP.

5.3.1. Pruebas realizadas

En el ejemplo de la captura Wireshark de la figura 5.11 utilizamos una matriz de 5 filas por 8 columnas, y cambiamos el grupo multicast de destino de los paquetes FEC por el 224.50.50.50 y el 224.10.10.20 para comprobar que se enviaban 5 paquetes por las 5 filas y 8 paquetes por las 8 columnas con un total de 13 paquetes de recuperación para un número de 5x8=40 paquetes originales.

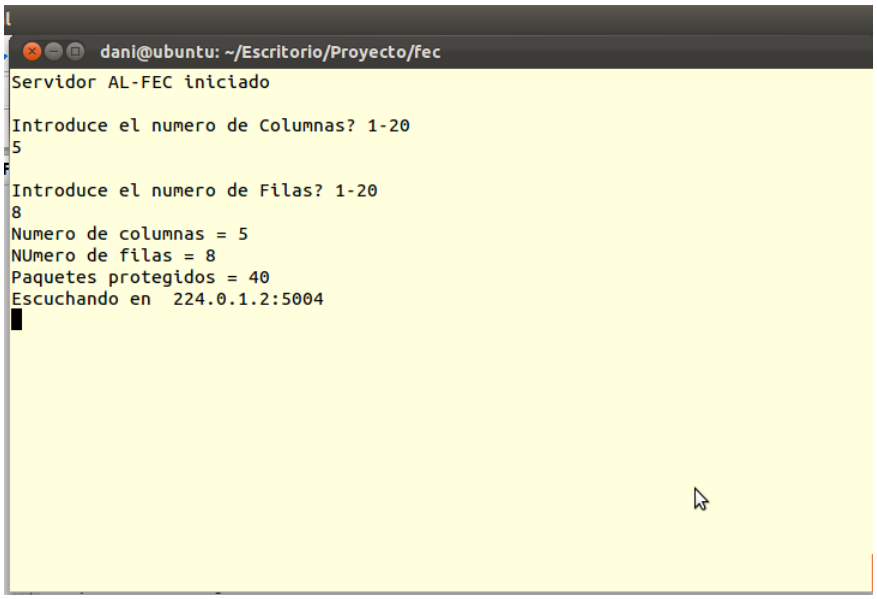

Fig 5.11 Servidor FEC iniciado, protegiendo 40 paquetes.

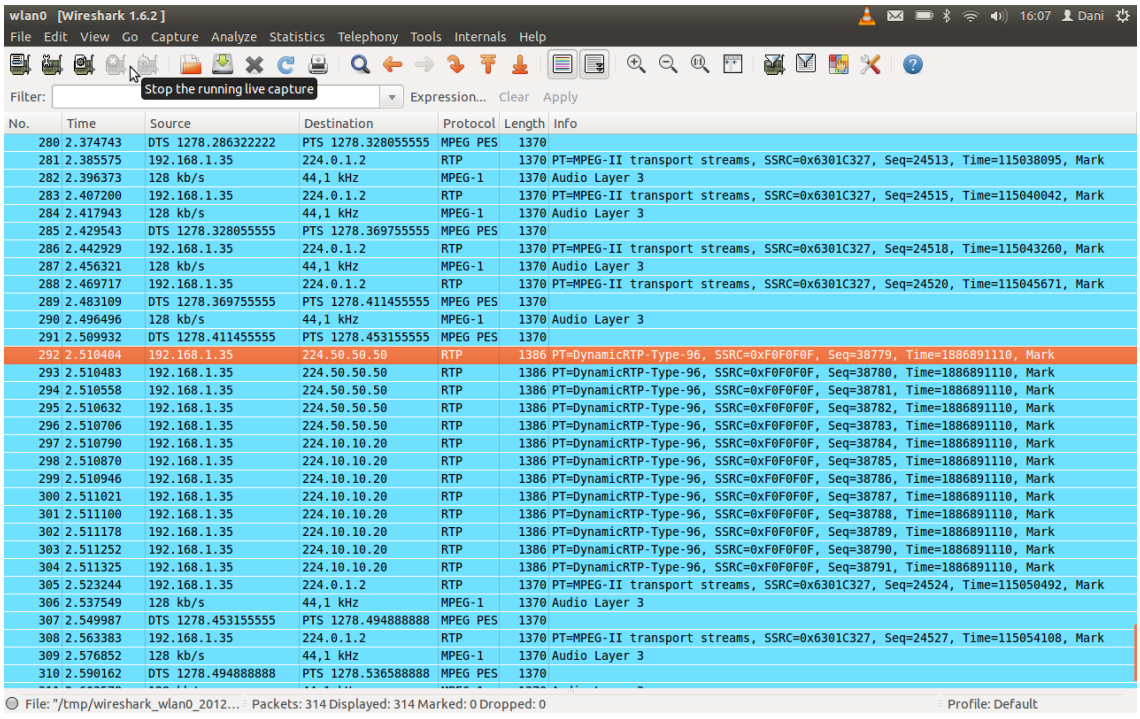

Fig 5.12 Captura de Wireshark capturando paquetes de corrección.

En la figura 5.12 podemos observar los 13 paquetes FEC protegidos, con un payload type que difiere de los paquetes RTP comunes, concretamente es Dynamic RTP type 96. Podemos ver como los números de secuencia de los paquetes FE también son diferentes, ya que llevan su orden particular. Los paquetes FEC que aparecen en la imagen son los que protegen a los paquetes RTP anteriores, concretamente los 40 anteriores.

Conclusiones y líneas futuras

El propósito de este proyecto era realizar un estudio general del protocolo DVB-IP y luego centrarnos en el protocolo RTSP y en la parte de protección de datos, implementando un servidor de retransmisiones RTP con su cliente y un servidor AL-FEC.

La primera fase del proyecto consistió en la documentación sobre la transmisión de video, estudiando la codificación, multiplexación y transmisión de video. El siguiente paso fue comenzar con el protocolo DVB-IP, consultando los diferentes estándares que han desarrollado y los diferentes trabajos final de carrera que otros estudiantes realizaron en su día.

Una vez finalizada la formación comenzamos con la fase de testeo de diferentes capturadoras de TV digital que disponíamos en el laboratorio, haciendo uso de las antenas que I2CAT tiene instaladas en la azotea del edificio. Se realizaron pruebas con dvblast, vlma y multicat (desarrollados por el proyecto VideoLAN), dvbstram, live555 y mumuDVB. Después de testear las diferentes posibilidades optamos por incorporar a nuestro escenario a dvblast para realizar el streaming y Live 555 para el contenido bajo demanda. Vlma nos proporcionaba muchas posibilidades para abarcar todo un escenario DVB-IP por si sola, pero estaba en una fase beta con lo cual había problemas con las tarjetas receptoras y con la transmisión de video bajo demanda.

La siguiente fase fue comenzar a estudiar la estructura de nuestro servidor de retransmisiones adaptándonos a lo que marca el estándar. Nos basamos en una aplicación llamada dumprtp [49] cuya única función era conectarse a grupos multicast y recibir streaming de video. A partir de aquí comenzamos a diseñar un servidor capaz de almacenar paquetes RTP con el fin de poder retransmitirlos en caso de pérdidas a los clientes que lo soliciten. A la vez fuimos diseñando un cliente capaz de controlar los paquetes recibidos comprobando que los números de secuencia seguían el orden correcto, y que en caso de pérdidas de paquetes fuese capaz de enviar una petición al servidor para que éste lo retransmitiera.

Cuando el servidor y cliente de retransmisión comenzaban a funcionar de manera óptima tuvimos que pensar en cómo ocasionar pérdidas en nuestro escenario, y decidimos diseñar una pequeña aplicación, utilizando funciones presentes en el servidor/cliente de retransmisión, que fuese capaz de conectarse al flujo de datos y reenviar solo una parte de él, provocando pérdidas.

Una vez teníamos el servidor y cliente de retransmisiones nos centramos en el desarrollo del servidor AL-FEC. Hemos trabajado a partir de un código que diseñó un estudiante anterior. Realizó una versión inicial de AL-FEC limitada a una dimensión, es decir, un paquete de corrección por cada N paquetes originales, mediante la operación XOR bit a bit sobre los paquetes RTP. Nosotros hemos mejorado el código para que realice la operación XOR bit a bit

en un matriz bidimensional, con lo que conseguimos una protección mayor en caso de ráfagas de pérdidas.

El siguiente paso ha sido realizar un estudio del protocolo RTSP. Dicho protocolo es específico de streaming y da soluciones para escenarios DVB-IP. Para ello hemos trabajado con un conjunto de aplicaciones de Live 555, éstas ofrecen un sinfín de posibilidades para realizar pruebas utilizando el protocolo RTSP. A la vez hemos comparado lo que dicta el estándar de la IETF [8] y la organización DVB [9] a nivel de métodos y cabeceras para los diálogos entre un cliente y un servidor RTSP y la aplicación Live Media Server, que hemos utilizado para realizar el video bajo demanda en nuestro escenario RTP.

En resumen, se dispone de:

- Un cliente de streaming con la capacidad de pedir retransmisiones, diseñado a partir del estándar.

- Un servidor de streaming capaz de atender peticiones de retransmisiones.

- Un servidor de streaming secundario capaz de provocar pérdidas de paquetes.

-Un servidor AL-FEC que realiza la protección de los datos aplicando la operación binaria XOR en una matriz bidimensional.

- Un estudio del protocolo RTSP para futuras implementaciones de un cliente y un servidor RTSP basado en el estándar.

Repercusión medioambiental

Las repercusiones medioambientales que puedan estar ocasionadas por este proyecto pueden basarse en el gasto eléctrico que puede ocasionar la refrigeración de una sala de servidores, el uso de los equipos particulares por parte de los usuarios, así como la fabricación de éstos.

Se podría contemplar la posibilidad que en un futuro las transmisiones de TV digital dejasen de emitirse por radiofrecuencia y se implementasen por la red de redes (Internet), limpiando el aire de contaminación electromagnética provocada por las miles de emisiones que se efectúan, tanto por antenas terrestres como por antenas satélites.

Aspectos éticos

No se aprecia ninguna implicación ética en este proyecto que podamos destacar. Podemos plantear el contenido de las diferentes emisiones, que implicaran visiones para adultos no adecuadas para menores de edad, manifestaciones políticas o religiosas, aunque creemos que en la implicación

ética que concierne a nuestro proyecto es la misma que se puede plantear en las emisiones de TV digital actuales.

Aspectos de Seguridad

En nuestro proyecto se plantean los aspectos se seguridad relevantes a las pérdidas de datos provocadas por el medio, no se valoran los aspectos de seguridad de accesos no legítimos ni alteración de los contenidos.

En nuestros servidores se podría plantear la posibilidad de utilizar firewalls para evitar accesos fraudulentos que pudiesen modificar la funcionalidad de éstos.

Si en un futuro estos tipos de escenarios han crecido por la red, se debería pensar en sistemas de Seguridad para controlar el acceso a contenidos, y limitar la emisión de éstos, para asegurarnos de que todo se adecua a la legalidad.

Líneas futuras

En un futuro podemos pensar en el diseño de un cliente que funcione para nuestro servidor AL-FEC con su matriz bidimensional, ya que podemos trabajar a partir del servidor ya diseñado. A partir de esta memoria se puede entender como se realiza la codificación de los diferentes campos del paquete RTP con la operación binaria XOR y así diseñar un nuevo cliente capaz de reconstruir los datos a partir de los paquetes FEC enviados por el servidor. También se podría plantear la posibilidad de crear un servidor conjunto que fuese capaz de implementar los dos sistemas de protección contra pérdidas de paquetes que hemos desarrollado, retransmisiones RTP y AL-FEC.

Se deberían realizar pruebas más sistemáticas de evaluación del rendimiento de los dos sistemas de protección, intentando generar patrones de pérdidas similares a los de una red IP real.

En el transcurso de los años diferentes estudiantes han trabajado en el diseño de diferentes partes de un escenario DVB-IP, como pueden ser un servidor DNS que resuelva direcciones IP, un servidor DHCP que asigne direcciones, un servidor que sea capaz del descubrimiento de los servicios junto a clientes que sean capaces de entender los documentos XML y averiguar los servicios ofrecidos, servidores para la protección de los datos y una la guía interactiva para los servicios de contenido bajo demanda. Todas las partes del escenario podían unirse para formar un escenario completo de DVB-IP, con un servidor que sea capaz de realizar todos los servicios, y un cliente capaz de recibir los contenidos y comprobar que todos los datos son correctos. A partir de aquí, ya dispondremos de un escenario completo de TV por IP, basándonos en lo establecido por el estándar DVB-IP.

Bibliografía

[1] F. Granaiola, “Development and Deployment of a DVB-IP Scenario”. Thesias of Laurea Specialistica of Telecommunication Engineering. University of Pisa / Universitat Politècnica de Catalunya, April 2010.

[2] DVB BlueBook A86: " Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IP Phase 1 specifications, ETSI ".

[3] , ETSI TS 102 542 V1.2.1: " Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IP Phase 1 specifications ".

[4] ETSI TS 102 542-1 V1.3.2 : " Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IP Phase 1 specifications. Part1: Core IPTV Functions ".

[5] ETSI TS 101 154 v1.10.1 Digital Video Broadcasting (DVB); Specification for the use of Video and Audio Coding in Broadcasting Applications based on the MPEG-2 Transport Stream”.

[6] ETSI TS 102 542-2: ”Digital Video Broadcasting (DVB); Guidelines for the implementation of DVB-IPTV Phase 1 specifications; Part 2: Broadband Content Guide (BCG) and Content on Demand”.

[7] RFC 2782 “A DNS RR for specifying the locationa of services (DNS SRV)”.

[8] RFC 2326 “Real Time Streaming Protocol 2.0 (RTSP)”.

[9] ETSI TS 102 034 V1.4.1: “Digital Video Broadcasting (DVB); Transport of MPEG-2 TS Based DVB Services over IP Based Networks.

[10] ETSI TS 102 826 V1.2.1: “DVB-IPTV Profiles for TS 102 034”.

[11] ETSI TS 102 542-3-3 V1.3.1: “Guidelines for the implementation of DVB-IP Phase 1 specifications. Part 3: Error Recovery. Subpart 3: Retransmission”.

[12] ETSI TS 102 542-3-2 V1.3.2 : “Guidelines for the implementation of DVB-IP Phase 1 specifications. Part 3: Error Recovery. Subpart 2: Application Layer FEC”

[13] Artículo JITEL 2010 D. Rincón, F. Granaiola, I. Rodríguez, “Desarrollo y despliegue de servicios DVB-IP con software open source”.

[14] RFC 4588 RTP Retransmission Payload Format

[15] RFC 6683 Guidelines for Implementing Digital Video Broadcasting - IPTV (DVB-IPTV) Application-Layer Hybrid Forward Error Correction (FEC) Protection

[45] D. Rincón, A. Oller, Curso DVB-IP. EPSC-UPC, Barcelona, 2008.

[46] Eugenio Viudez “Desarrollo de un cliente DVB-IP con perfil Live Media Broadcast (LMB)” Universistat Politécnica de Catalunya.

[47] SMPTE “Forward Error Correction for Real-Time Video/Audio Transport Over IP Networks”
http://standards.smpte.org/content/978-1-61482-615-6/st-2022-1-2007/SEC1.abstract?sid=3b9a71f5-1c05-4997-892c-67b9662070a1

[48] Real-Time Transport Protocol (RTP) Parameters - IANA
http://www.iana.org/assignments/rtp-parameters

Programación

[16] Principios básicos de la programación en C.
http://www.monografias.com/trabajos33/programacion-lenguaje-c/programacion-lenguaje-c.shtml

[17] Procesos en C
http://www.chuidiang.com/clinux/procesos/procesoshilos.php#procesoshilos

[18] Operadores en C
http://es.wikipedia.org/wiki/Operadores_de_C_y_C%2B%2B

[19] Punteros en C
http://www.it.uc3m.es/labas/DSP/M2/Pointers_es.html

[20] Información sobre C
http://html.rincondelvago.com/programacion-en-c_6.html

[21] Ejemplos de threads en C
http://www.ib.cnea.gov.ar/~servos/Practicas/thread.html

Herramientas utilizadas

[22] Wireshark.
http://www.wireshark.org

[23] Notepad++
http://notepad-plus.sourceforge.net/es/site.htm

[24] VideoLan
http://www.videolan.org/

[25] Live555
http://www.live555.com/

[26] Dvbstream
http://www.linuxtv.org/wiki/index.php/Dvbstream

[27] Dvblast
http://www.videolan.org/projects/dvblast.html

[28] Vlma
http://www.videolan.org/projects/vlma/

[29] MumuDvb
http://mumudvb.braice.net/mumudrupal/

[30] CodeBlocks
http://www.codeblocks.org

[49] Dumprtp
http://sourcecodebrowser.com/dvbstream/0.5/dumprtp_8c.html

Varios

[31] Wikepdia
http://es.wikipedia.org/wiki/Wikipedia:Portada

[32] European Telecommunications Standards Institute (ETSI).
http://www.etsi.org

[33] Linux TV
http://www.linuxtv.org

[34] Organización Dvb
www.dvb.org

[35] Canales terrestres
http://www.tdt1.com/

[36] Canales Satélite
http://es.kingofsat.net/search.php

[37] Ubuntu
http://www.ubuntu.com/

[38] Scripts en Ubuntu
http://www.ucm.es/info/aulasun/archivos/SCRIPTS.pdf

[39] Imagenio
http://www.movistar.com/Imagenio

[40] Orange
http://www.orange.com

[41] Jazztel
http://www.jazztel.com

[42] Youtube
http://www.youtube.com

[43] Kaffeine
http://www.kaffeine.com

[44] SMPTE
http://www.smpte.com

Glosario

A continuación se recopilan las abreviaturas utilizadas en este documento:

ADSL	Asymmetric Digital Subscriber Line
AL-FEC	Application Layer-Forward Error Correction
BCG	Broadband Content Guide
CDS	Content Download Service
CoD	Content on Demand
DHCP	Dynamic Host Configuration Protocol
DNG	Delivery Network Gateway
DOM	Document Object Model
DVB	Digital Video Broadcasting
DVBSTP	DVB SD&S Transport Protocol
EPG	Electronic Program Guide
HNED	Home Network End Device
HTTP	Hyper Text Transfer Protocol
IGMP	Internet Group Management Protocol
IP	Internet Protocol
IPTV	IP Television
ISP	Internet Service Provider
LMB	Live Media Broadcast
MPEG	Moving Pictures Expert Group
MPEG-2 TS	MPEG-2 Transport Stream
MTU	Maximum Transfer Unit
NTP	Network Time Protocol
RTSP	Real Time Streaming Protocol
SD&S	Service Discovery and Selection
SI	Service Information
SIP	Session Initiation Protocol
SP	Service Provider
TS	Transport Stream
TVA	TV-Anytime
UDP	User Datagram Protocol
URI	Uniform resource identifier
URL	Uniform Resource Locator
XML	eXtensible Markup Language

Anexos

Servidor de Retransmisión

A continuación el código diseñado para crear un servidor de retransmisión que sea capaz de escuchar por dos puertos simultáneos utilizando threads. Por un puerto escuchará el flujo normal de paquetes RTP y por el otro atenderá peticiones de retransmisión.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <resolv.h>
#include <unistd.h>
#include "rtp_serv.h"
#include "rtp_serv.c"
#include <pthread.h>

//variable global para almacenar un paquete
char almacena[1000][1327];
//guardaremos la medida de cada paquete almacenado
char size_rtp[10];
//Mutex para serializar el acceso a variables globales
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;


void *atenderpeticiones(void *parametro){

   //VAriable para almacena recorrer
    int i,j;
    //VAriable para guardar el numero de secuencia que me piden
   int peticion[1];
   int sockfd,n;
   struct sockaddr_in servaddr,cliaddr;
   socklen_t len;
   int datos_recibidos[1];
   char datos_recibidos2[2];
   char datos_enviar[1327];

   char *ip_ret;
   char *ip_ret2;
   int port_ret;
   int port_ret2;
   ip_ret  = "224.100.100.100";    port_ret = 50000;

   pthread_t idHilo2;

   struct sockaddr_in ret;
   struct sockaddr_in ret2;
   int socketRET;
   int socketRET2;

   socketRET  = makeclientsocket(ip_ret,port_ret,2/*ttl*/,&ret);
   socketRET2  = makeclientsocket(ip_ret,port_ret+2,2/*ttl*/,&ret2);
```

```
   char numero_sec[2];
   int q;
   char secuencia_alamcena[2];
   struct packet_RTCPFB fb;
   int paquete_pedido;

   //Bucle para escuchar peticiones de retransmision
   for (;;)
   {
     paquete_pedido= getrtp_feedback(socketRET, &fb);

     printf("           El paquete perdido es: %d", paquete_pedido);
    //Si recibo algun dato tengo que bloquear el acceso a la variable
global
     pthread_mutex_lock(&mymutex);
      //Recorremos la matriz almacena y guardamos los numeros de
secuencia para luego comparar con los datos recibidos.
      //LA funcion secuencia nos devuelve un entero (1 byte) que viene
a ser el numero de secuencia, a esta funcion se le pasa un 2 bytes
donde viene el numero de secuencia en la cabecera de un paquete RTP
      for(i=0;i<100;i++){
          secuencia_alamcena[0]=almacena[i][2];
          secuencia_alamcena[1]=almacena[i][3];

          if (paquete_pedido==secuencia(secuencia_alamcena)){

            //Copiamos el paquete para enviarlo
            for(j=0; j<1327; j++){
                datos_enviar[j]=almacena[i][j];
            }
            //Le enviamos el paquete que nos ha pedido
            sendto(socketRET,datos_enviar,1328,0,(struct       sockaddr
*)&ret2,sizeof(ret2));
            printf("\n             Paquete %d enviado al usuario\n\n",
paquete_pedido);
          i=100;
          }
          //NO tenemos el paquete solicitado
          else{

            //sendto(socketRET2,datos_enviar,1590,0,(struct    sockaddr
*)&ret2,sizeof(ret2));
          }

      }
      //DEsbloqueamos el acceso a la variable global
      pthread_mutex_unlock(&mymutex);

   }
pthread_exit((void*)"Acabe");
}
void dumprtp( int socket, struct sockaddr_in k) {
  char* buf;
  struct rtpheader rh;

  int lengthData;
  unsigned short seq=0;
  int len=0;
  //MIS VARIABLES
  int c;
```

```
  char numero_sec[2];
  int lon_Rtp=0;
  int lon_RET=10;
  //Variables para almacenar en buffer
  int i=0,j=0; //La i tambien sirve para guardar numero de secuencia
se encuentra en la fila correcta
  //Variables para borrar almacenar
  int q=0,z=0;
  //Variables para guardar numero de secuencia
  int x=2;

  int cont=0,g=0,h,a=0;

  char datos[1327];

  pthread_t idHilo;
  char *valorDevuelto =NULL;
  int error;

  //El thread que creeamos para atender peticiones de Un cliente
  error = pthread_create(&idHilo, NULL, atenderpeticiones,NULL);

  if(error!=0){
        perror("No puedo crear el hilo de ejecución");
        exit(-1);
  }
   for(;;)
   {

   //RECIBIR LOS DATOS
   lon_Rtp=recv(socket,datos,1590,0);
    //Copio los datos recibidos en un array de 10filas y 1500 olumnas
    for(j=0; j<lon_Rtp;j++){
            almacena [i][j] = datos[j];
            size_rtp[i] = lon_Rtp;//Gurdamos el tamañño del paquete
porque no siempre seran de 1328;
    }

    //COmenzamos a guardar de nuevo
    if(i==1000){
    i=0;
    }

    for(q=0; q<2; q++){
         numero_sec[q]=almacena[i][x];
         x++;
    }
   printar(numero_sec);
    //Siempre tiene que ser el 3 byte, donde empieza el numero de
secuencia
    x=2;
a++;
i++;//Cada vuelta al bucle un paquete nuevo una fila de la array


}


//pthread_join (idHilo, (void **)&valorDevuelto);
```

```
exit(0);
}

int main(int argc, char *argv[]) {

  system("clear");
  struct sockaddr_in si;
  int socketIn;
  int socketIn2;

  char *ip;
  char *ip2;

  int port;
  int port2;

  fprintf(stderr,"\nServidor de retransmision inicidao\n");

  if (argc == 1) {
    ip   = "224.0.1.2";    port = 5004;

  }
  else if (argc == 3) {
    ip   = argv[1];
    port = atoi(argv[2]);
  }
  else {
    fprintf(stderr,"Usage %s ip port\n",argv[0]);
    exit(1);
  }

  fprintf(stderr,"Emision de video en %s:%d\n",ip,port);
  fprintf(stderr,"Esuchar peticiones RET 224.100.100.100:50000\n");



  socketIn  = makeclientsocket(ip,port,2/*ttl*/,&si);


  dumprtp(socketIn,si);
  close(socketIn);
  return(0);

}
```

Cliente de retransmisión

A continuación el código diseñado para un cliente capaz de pedir retransmisiones en caso de pérdida de paquetes.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <resolv.h>
#include <unistd.h>
#include <sys/time.h>
```

```
    #include <strings.h>
    #include <errno.h>
    #include "rtp_cli.h"
    #include "rtp_cli.c"
    #include <pthread.h>

//VAriables globales
char almacena[1000][1500];
pthread_mutex_t mymutex=PTHREAD_MUTEX_INITIALIZER;


void *enviar_peticion_RET(void *parametro){    //Le pasamos un string
con el numero de paquete perdido y con la posicion en la matriz
   int sockfd,n;
   //Creamos un puntero para pasar los datos del string a una posicion
de memoria
   int* variables;
   variables = (int*)parametro;
   int posi_guardar;
   int h;
 //Copiamos el primer byte (numero de secuenca), y luego la el segundo
byte (posicion en la matriz)
   int  paquete_que_necesito[50];

   for(h=0;h<50;h++){paquete_que_necesito[h]=1;}

   memcpy(&posi_guardar,variables, sizeof(int)); //Posicion del primer
paquete perdido

   int p;
   int paquetes_perdidos=0;
   for(p=0;p<50;p++){

        if(paquete_que_necesito[p-1]=='\0'){
            paquetes_perdidos = p-1;
            p=50;

        }
        else{
            memcpy(&paquete_que_necesito[p],variables+p+1,
sizeof(int));

        }
   }

   struct sockaddr_in servaddr,cliaddr;
  // char sendline[1000];
   char recvline[1500];
   char paquete_RET[1327];

   int long_recibida;
   char numero_secuencia_recibido[2];
   //VAriables para copiar en la matriz
   int k;

   //Estructura cabecera RTCP feedback
   struct packet_RTCPFB fb;

   int paquete;
   int paquete_que_voy_a_pedir[1];
    int numero_paquete_escuchado;
```

```
   char numero_paquete_escuchado_RET[2];

   char *ip_ret;
   char *ip_ret2;
   int port_ret;
   int port_ret2;

   ip_ret  = "224.100.100.100";    port_ret = 50000;
   ip_ret2 = "224.150.150.150";     port_ret2=60000;


   struct sockaddr_in ret;
   struct sockaddr_in ret2;
   int socketRET;
   int socketRET2;

   socketRET  = makeclientsocket(ip_ret,port_ret,2/*ttl*/,&ret);
   socketRET2  = makeclientsocket(ip_ret,port_ret+2,2/*ttl*/,&ret2);

  //Tenemos que escuchar un tiempo para enviar la peticion de RET, ya
que otro cliente ha podido perder el mismo paquete

  struct timeval tv;
  //Tiempo de espera sera entre 0 y 2 segundos
  tv.tv_sec=rand()%6;

  tv.tv_usec=0;

  n   =   setsockopt(   socketRET,   SOL_SOCKET,   SO_RCVTIMEO,   &tv,
sizeof(tv));

  do {
  //Esperamos 5 segundos para comprobar si hay alguna peticion de RET
o hay alguna RET en movimiento que nos sirva

  n = recv(socketRET,recvline,1500,0);
  if (n<0)
  {
      if (errno == EWOULDBLOCK){
        printf("\n\n     Se ha agotado el tiempo de escucha procedemos
a enviar peticiones\n");
      }
      else
      {
        printf("Error en el recvfrom\n");
      }
  }

  else{
    printf("\n  Ha llegado algun paquete un paquete\n");

    if (n>1000)//Hemos recibido un paquete retransmitido
    {
        numero_secuencia_recibido[0] = recvline[2];
        numero_secuencia_recibido[1] = recvline[3];
        numero_paquete_escuchado=
secuencia(numero_secuencia_recibido);
        for(p=0;p<paquetes_perdidos;p++){

           if(paquete_que_necesito[p]  ==  numero_paquete_escuchado)//El
paquete recibido es un paquete que he perdido
```

```
            {
                  pthread_mutex_lock(&mymutex);

                  for(k=0; k<1500; k++){
                      almacena[posi_guardar+p][k]=recvline[k];
                  }
                  //desbloqueamos el acceso a la variable global

                  pthread_mutex_unlock(&mymutex);
            }

        }

    }
    else
    {
        //EL PAQUETE ES UNA PETICION DE RET
        numero_paquete_escuchado_RET[0]=recvline[12];
        numero_paquete_escuchado_RET[1]=recvline[13];
        numero_paquete_escuchado                                    =
secuencia(numero_paquete_escuchado_RET);
        printf("\n   Hemos escuchado una petición de retransmisión,
procedemos a escuchar para recibir la retransmisión\n");
        for(p=0;p<paquetes_perdidos;p++){

          if(paquete_que_necesito[p]  ==  numero_paquete_escuchado)//El
paquete que pide el otro cliente no lo voy a pedir
            {
               paquete_que_necesito[p]='\0';
               paquetes_perdidos--;
            }
        }
        //NOs ponemos a escuchar para recibir el paquete que ha pedido
el otro cliente
        long_recibida=recv(socketRET2,recvline,1500,0);
        numero_secuencia_recibido[0] = recvline[2];
        numero_secuencia_recibido[1] = recvline[3];
        numero_paquete_escuchado=
secuencia(numero_secuencia_recibido);
        printf("HEmos recibido algo, paquete retransmitido NumSec=
%d", numero_paquete_escuchado);
        for(p=0;p<paquetes_perdidos;p++){

           if(paquete_que_necesito[p] == numero_paquete_escuchado)//El
paquete recibido es un paquete que he perdido
            {
                  pthread_mutex_lock(&mymutex);

                  for(k=0; k<1500; k++){
                      almacena[posi_guardar+p][k]=recvline[k];
                  }
                  //desbloqueamos el acceso a la variable global

                  pthread_mutex_unlock(&mymutex);
            }
        }

        }
 }

} while(errno=!EWOULDBLOCK);
```

```
 //Haremos tantas peticiones como paquetes hemos perdido
  for(p=0;p<paquetes_perdidos;p++){

       paquete = paquete_que_voy_a_pedir[0];

       initrtp_feedback(&fb, paquete_que_necesito[p]);

       sendrtp_feedback(socketRET, &ret, &fb);

       printf("        \nPeticion  RET  enviada,  esperando  recibir  el
paquete\n");

       long_recibida=recv(socketRET2,recvline,1500,0);

       if (long_recibida<10){
         printf("\n        No  hemos  conseguido  recuperar  el  paquete
perdido\n");
       }
       else{
         //Ahora toca guardarlo en su posicion
         //Bloqueamos el acceso a la variable global
         numero_secuencia_recibido[0] = recvline[2];
         numero_secuencia_recibido[1] = recvline[3];
         if
(paquete_que_necesito[p]==secuencia(numero_secuencia_recibido))
         {

             printf("\n                        Paquete      Retransmitido:
%d\n",secuencia(numero_secuencia_recibido));
             pthread_mutex_lock(&mymutex);



                 for(k=0; k<1500; k++){
                     almacena[posi_guardar+p][k]=recvline[k];
             }
             //desbloqueamos el acceso a la variable global
             pthread_mutex_unlock(&mymutex);

         }

          else {

          printf("Hemos recibido un paquete que no queríamos");
          }
        }
      }
     pthread_exit((void*)"He acabado");

}
void dumprtp( int socket, struct sockaddr_in k) {

  char* buf;
  struct rtpheader rh;
  int lengthData;
  unsigned short seq=0;
  int len=0;

  //MIS VARIABLES
  int c, p;
```

```
  int lon_Rtp=0;
  int lon_RET=10;
  //Variables para almacenar en buffer
  int i=0,j=0; //La i tambien sirve para guardar numero de secuencia
se encuentra en la fila correcta
  int cont_adelantado = 0;
  //Variables para borrar almacenar
  int q=0,z=0;
  //Variables para guardar numero de secuencia
  int x=2;
  //Variable bandera para saber que hemos empezado
  int primera_vuelta = 0;
  int cont=0,g=0,h,a=0;
  //Variables contabilizar paquetes
  int                                          numero_ultimo_paquete=0,
numero_ultimo_paquete_guardado=0,paquetes_faltan=0;
  //Variables almacenar numeros de secuencia
  char numero_sec[2];
  char numero_sec_ant[2];
  //Donde se vuelcan los datos recibidos
  char datos[1327];

  //Variable para crear thread
  pthread_t idHilo;

  char *valorDevuelto =NULL;
  int error;
 //VAriables de retransmision
  int posi_bufer[5]; //Posicion donde se guardara el paquete perdido
  int paquete_peticion=0;
  //int envios = 0;//Numero de veces que ejecutaremos la funcion de
pedir peticion de retransmsion en funcionde  los paquetes enviados
  //VAriables para pasarle datos al thread
  int datos_func_thread[100];


int paquete_a_pedir[50];
int posiciones_paquetes[50];


   while(a<100){

    //RECIBIR LOS DATOS
    lon_Rtp=recv(socket,datos,1590,0);
   // printf("He recibido: %d", lon_Rtp);
    //COMPROBACION DE LOS DATOS QUE HE RECIBIDO Printar Cabecera
   // getNuevo(datos,&rh,&buf,&len);
    numero_sec[0]=datos[2];
    numero_sec[1]=datos[3];

    //Comprobamos que el paquete anterior, funcion secuencia devuelve
un entero
    printf("\nPaquete recibido NumSec=%d",secuencia(numero_sec));
    //Comparamos el paquete recibido con el ultimos guardado
    //Primera vuelta es un semaforo para entrar por primera vez cuando
recibamos el primer paquete
    //El paquete recibido tiene que ser el anterior + 1
    if ((secuencia(numero_sec)==numero_ultimo_paquete_guardado +1 ) ||
primera_vuelta == 0){
```

```
        //Copio los datos recibidos en un array de 10filas y
1328columnas

        for(j=0; j<lon_Rtp;j++){
            almacena [i][j] = datos[j];
            //Guardo el numero desecuencia

        }

        numero_sec_ant[0] = datos[2];
        numero_sec_ant[1] = datos[3];
        //printar(numero_sec_ant);
        a++;//COntador provisional para el while
        i++;//Cada vuelta al bucle un paquete nuevo una fila de la
array
        //Una vez damos la primra vuelta la bandera se pone a 1 y la
comprobación con el paquete anterior se realiza
        primera_vuelta = 1;
        numero_ultimo_paquete_guardado = secuencia(numero_sec);


    }
    //Nos ha faltado un paquete
    else
    {
         printf("\n    Algun paquete se ha perdido\n");
         //Guardo el numero de secuencia
         numero_ultimo_paquete = secuencia(numero_sec);
         printf("           Paquete recibido:%d - Ultimo paquete
guardad:%d\n",numero_ultimo_paquete,numero_ultimo_paquete_guardado);
         numero_ultimo_paquete_guardado = secuencia(numero_sec_ant);
         //SABER LOS PAQUETES QUE NOS FALTAN
         paquetes_faltan        =        numero_ultimo_paquete        -
numero_ultimo_paquete_guardado;
         paquetes_faltan = paquetes_faltan -1;
         printf("    Paquetes perdidos = %d",paquetes_faltan);

        //Almacenamos el ultimo paquete que nos ha llegado
        cont_adelantado = i + paquetes_faltan;
                 j=0;
                 for(j=0; j<1500;j++){
                     almacena [cont_adelantado][j] = datos[j];


                 }
         //Guardamos la posicion del primer paquete que nos falta
         datos_func_thread[0] = i;
         //Con este for almacenamos los numeros de secuencia que nos
han faltado
         for (p=1;p<paquetes_faltan+1;p++){

             datos_func_thread[p] = numero_ultimo_paquete_guardado + p;
                 }
         //Ponemos una bandera para saber cual es el final del vector
         datos_func_thread[p]='\0';
         //Creamos un hijo y le pasamos la variable a la funcion del
thread
         error          =          pthread_create(&idHilo,          NULL,
enviar_peticion_RET,datos_func_thread);

                 if(error!=0){
```

```
                    perror("No puedo crear el hilo de ejecución");
                exit(-1);
        }
       pthread_join (idHilo, (void **)&valorDevuelto);
        i = i + paquetes_faltan +1;
        numero_ultimo_paquete_guardado=secuencia(numero_sec);

   }

a++;


}


//Printar los numeros de secuencia que he guardado
int ultim_cont = 0;
char paquetito[2];


pthread_join (idHilo, (void **)&valorDevuelto);

//Printo todo lo que tengo guardado para ver si realmente me han
retransmitido todo lo que necesitaba
for(ultim_cont = 0 ; ultim_cont<20 ; ultim_cont++)
{
    paquetito[0]=almacena[ultim_cont][2];
    paquetito[1]=almacena[ultim_cont][3];

    printf("\nPaquete guardado numero: %d\n", secuencia(paquetito));

}
exit(0);
}

int main(int argc, char *argv[]) {
  system("clear");
  struct sockaddr_in si;
  int socketIn;
  int socketIn2;

  char *ip;
  char *ip2;

  int port;
  int port2;


  if (argc == 1) {
    ip = "224.50.50.50";
    port = 15004;

  }
  else if (argc == 3) {
    ip   = argv[1];
    port = atoi(argv[2]);
  }
  else {
    fprintf(stderr,"Usage %s ip port\n",argv[0]);
    exit(1);
  }
```

```
  fprintf(stderr,"Cliente RTP \nEscuchando en: %s:%d\n",ip,port);

  socketIn  = makeclientsocket(ip,port,2/*ttl*/,&si);

  dumprtp(socketIn,si);
  close(socketIn);
  return(0);
}
```

Funciones utilizadas para las aplicaciones de retransmisión

Esta función inicializa la cabecera de un paquete RTP

```
void initrtp(struct rtpheader *foo,int pt, int type) { /* fill in the
  foo->b.v=2;
  foo->b.p=0;
  foo->b.x=0;
  foo->b.cc=0;
  foo->b.m=0;
  foo->b.pt=pt;
  foo->b.sequence=rand() & 65535;
  foo->timestamp=rand();
  foo->ssrc=rand();
  foo->type = type;
}
```

Esta función inicia la cabecera del paquete RTP Feedback utilizado para las peticiones de retransmisión

```
void initrtp_feedback(struct packet_RTCPFB *foo_fb, /*int ssrc,*/ int
SN_lost) {
 foo_fb->v=2;
 foo_fb->p=0;
 foo_fb->FMT=1;
 foo_fb->PT=205;             //for generic NACK message
 foo_fb->length=128;
 foo_fb->ssrc_psender=0;
 foo_fb->ssrc_msource=0;
 foo_fb->PID=SN_lost;
 foo_fb->BLP=0;


}
```

Esta función se utiliza en el servidor para recuperar los paquetes RTP Feedback enviados por los clientes.

```
int getrtp_feedback(int socket_ret, struct packet_RTCPFB *fb) {
```

```
 static char buf_fb[16];
// char *point = buf_fb;
 unsigned int intP;

 char* charP = (char*) &intP;
 int pcktfb;
 pcktfb = recv(socket_ret, buf_fb, 16, 0);
 if (pcktfb==0)
     exit(1);
 fb->v = (unsigned int) ((buf_fb[0]>>6)&0x03);
 fb->p = (unsigned int) ((buf_fb[0]>>5)&0x01);
 fb->FMT = (unsigned int) ((buf_fb[0]>>0)&0x1f);
 intP = 0;
 memcpy(charP,&buf_fb[1],1);
 fb->PT = ntohl(intP);
 intP = 0;
 memcpy(charP+2,&buf_fb[2],2);
 fb->length = ntohl(intP);
 intP = 0;
 memcpy(charP+4,&buf_fb[4],4);
 fb->ssrc_psender = ntohl(intP);
 intP = 0;
 memcpy(charP+4,&buf_fb[8],4);
 fb->ssrc_msource = ntohl(intP);
 intP = 0;
 memcpy(charP+2,&buf_fb[12],2);
 fb->PID = ntohl(intP);
 return fb->PID;
}
```

Función utilizada para enviar las peticiones de retransmisión RTCP Feedback.

```
int sendrtp_feedback(int socket_ret, struct sockaddr_in *sSockAddr,
struct packet_RTCPFB *foo_fb) {
 char num[2];
 char buf_fb[16];
 unsigned int intP;
 char *charP = (char*) &intP;

 buf_fb[0] = 0x00;
 buf_fb[0] |= ((((char) foo_fb->v)<<6)&0xc0);//1100 0000
 buf_fb[0] |= ((((char) foo_fb->p)<<5)&0x20);//0010 0000
 buf_fb[0] |= ((((char) foo_fb->FMT)<<0)&0x1f);//0001 1111


 intP = htonl(foo_fb->PT);
 memcpy(&buf_fb[1],charP,1);

 intP = htonl(foo_fb->length);
 memcpy(&buf_fb[2],charP+2,2);

 intP = htonl(foo_fb->ssrc_psender);
 memcpy(&buf_fb[4],charP+4,4);

 intP = htonl(foo_fb->ssrc_msource);
 memcpy(&buf_fb[8],charP+4,4);

 intP = htonl(foo_fb->PID);

 memcpy(&buf_fb[12],charP+2,2);
```

```
 intP = htonl(foo_fb->BLP);
 memcpy(&buf_fb[14],charP+2,2);

 num[0] = buf_fb[12];
 num[1] = buf_fb[13];


 return sendto(socket_ret, buf_fb, sizeof(buf_fb), 0, (struct sockaddr
*)sSockAddr, sizeof(*sSockAddr));
}
```

Función que que devuelve un número de secuencia a partir de una variable char.

```
int secuencia(char buf[2]) {

  unsigned int intP;
  char* charP = (char*) &intP;
  intP = 0;
  memcpy(charP+2,&buf[0],2);
  unsigned int numero;
  numero = ntohl(intP);
  intP = 0;


  return numero;
}
```

Función para crear un socket de recepción multicast

```
/* create a receiver socket, i.e. join the multicast group. */
int makeclientsocket(char *szAddr,unsigned short port,int TTL,struct
sockaddr_in *sSockAddr) {
  int socket=makesocket(szAddr,port,TTL,sSockAddr);
  struct ip_mreq blub;
  struct sockaddr_in sin;
  unsigned int tempaddr;
  sin.sin_family=AF_INET;
  sin.sin_port=htons(port);
  sin.sin_addr.s_addr=inet_addr(szAddr);
  if (bind(socket,(struct sockaddr *)&sin,sizeof(sin))) {
    perror("bind failed");
    exit(1);
  }
  tempaddr=inet_addr(szAddr);
  if ((ntohl(tempaddr) >> 28) == 0xe) {
    blub.imr_multiaddr.s_addr = inet_addr(szAddr);
    blub.imr_interface.s_addr = 0;
    if
(setsockopt(socket,IPPROTO_IP,IP_ADD_MEMBERSHIP,&blub,sizeof(blub))) {
      perror("setsockopt IP_ADD_MEMBERSHIP failed (multicast
kernel?)");
      exit(1);
    }
  }
  return socket;
}
```

Aplicación para provocar pérdidas

La siguiente aplicación es un servidor intermedio que provocará las pérdidas en un escenario DVB-IP. En este caso se perderán tres paquetes, dos de ellos consecutivos.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <resolv.h>
#include <unistd.h>

#include "rtp.h"
#include "rtp.c"

void dumprtp(int socket2, int socket, struct sockaddr_in k) {
  char* buf;
  struct rtpheader rh;
  int lengthData;
  unsigned short seq=0;
  int flag=0;
  int len=0;
  char datos[1370];
 int i =0 ;
  int headerRTP;
  int a=0;
  while(1) {

    getrtp2(socket,&rh,  &buf,&lengthData,  &headerRTP);  //it  catches
packets one by one

   if (a==5 || a==6 || a==9){
    }
    else {

        sendrtp2(socket2, &k, &rh, &buf,lengthData);
        }
  a++;
  }
}
int main(int argc, char *argv[]) {

  struct sockaddr_in si;
  int socketIn;
  int socketIn2;


  char *ip;
  char *ip2;


  int port;
  int port2;
  int port3;

  fprintf(stderr,"Rtp dump\n");

  if (argc == 1) {
    ip   = "224.0.1.2";
```

```
    ip2   = "224.50.50.50";

    port = 5004;
    port2 = 15004;

  }
  else if (argc == 3) {
    ip   = argv[1];
    port = atoi(argv[2]);
  }
  else {
    fprintf(stderr,"Usage %s ip port\n",argv[0]);
    exit(1);
  }

  fprintf(stderr,"Using %s:%d\n",ip,port);

  socketIn  = makeclientsocket(ip,port,2/*ttl*/,&si);
  socketIn2  = makeclientsocket(ip2,port2,2/*ttl*/,&si);


  dumprtp(socketIn2,socketIn,si);

  close(socketIn);

  return(0);
}
```

Funciones utilizadas para la aplicación de pérdidas

Función para recibir flujos RTP, analizando la cabecera previamente

```
int getrtp2(int fd, struct rtpheader *rh, char* data, int* lengthData)
{
  static char buf[1600];
  char *pointer = buf;
  unsigned int intP;
  char* charP = (char*) &intP;
  int headerSize;
  int lengthPacket;
  lengthPacket=recv(fd,buf,1328,0);
  if (lengthPacket==0)
    exit(1);
 if (lengthPacket<0) {
    //fprintf(stderr,"socket read error\n");
    return 0;
  }

  rh->b.v  = (unsigned int) ((buf[0]>>6)&0x03);
  rh->b.p  = (unsigned int) ((buf[0]>>5)&0x01);
  rh->b.x  = (unsigned int) ((buf[0]>>4)&0x01);
  rh->b.cc = (unsigned int) ((buf[0]>>0)&0x0f);
  rh->b.m  = (unsigned int) ((buf[1]>>7)&0x01);
  rh->b.pt = (unsigned int) ((buf[1]>>0)&0x7f);
  intP = 0;
  memcpy(charP+2,&buf[2],2);
  rh->b.sequence = ntohl(intP);
  intP = 0;
```

```
  memcpy(charP,&buf[4],4);
  rh->timestamp = ntohl(intP);

  headerSize = 12 + 4*rh->b.cc; /* in bytes */

  *lengthData = lengthPacket - headerSize;
   memcpy(data, pointer + headerSize,lengthPacket - headerSize);
  //*data = (char*) buf + headerSize;

  //   fprintf(stderr,"Reading rtp: v=%x p=%x x=%x cc=%x m=%x pt=%x
seq=%x  ts=%x  lgth=%d\n",rh->b.v,rh->b.p,rh->b.x,rh->b.cc,rh->b.m,rh-
>b.pt,rh->b.sequence,rh->timestamp,lengthPacket);

  return 1;
}
```

Función utilizada para enviar un paquete RTP formando su cabecera previamente.

```
int sendrtp2(int fd, struct sockaddr_in *sSockAddr, struct rtpheader
*foo, char *data, int len) {
  char *buf;
  unsigned int intP;
  char* charP = (char*) &intP;//serve per memorizzare il sequence
number: infatti ho bisogno di un puntatore
  int headerSize;
  if(foo->type == RTP) {
  buf=(char*)alloca(len+72);
  buf[0]  = 0x00;
  buf[0] |= ((((char) foo->b.v)<<6)&0xc0);
  buf[0] |= ((((char) foo->b.p)<<5)&0x20);
  buf[0] |= ((((char) foo->b.x)<<4)&0x10);
  buf[0] |= ((((char) foo->b.cc)<<0)&0x0f);
  buf[1]  = 0x00;
  buf[1] |= ((((char) foo->b.m)<<7)&0x80);
  buf[1] |= ((((char) foo->b.pt)<<0)&0x7f);
  intP = htonl(foo->b.sequence);
  memcpy(&buf[2],charP+2,2);
  intP = htonl(foo->timestamp);
  memcpy(&buf[4],&intP,4);
  /* SSRC: not implemented */
  buf[8]  = 0x0f;
  buf[9]  = 0x0f;
  buf[10] = 0x0f;
  buf[11] = 0x0f;
  headerSize = 12 + 4*foo->b.cc; /* in bytes */
  memcpy(buf+headerSize,data,len);

  //   fprintf(stderr,"Sending rtp: v=%x p=%x x=%x cc=%x m=%x pt=%x
seq=%x    ts=%x    lgth=%d\n",foo->b.v,foo->b.p,foo->b.x,foo->b.cc,foo-
>b.m,foo->b.pt,foo->b.sequence,foo->timestamp,len+headerSize);

  foo->b.sequence++;
  } else {  //UDP
    buf = data;
    headerSize = 0;
  }
  return        sendto(fd,buf,len+headerSize,0,(struct        sockaddr
*)sSockAddr,sizeof(*sSockAddr));
}
```

Servidor de AL-FEC

Aplicación modificada para el envío de paquetes FEC a un cliente. La aplicación original solo enviaba paquetes a partir de un vector XOR, ahora es capaz de enviar paquetes a partir de una matriz XOR.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <resolv.h>
#include <unistd.h>

#include "rtp.h"
#include "rtp.c"


#define MTU 1500
#define IP_HEADER_SIZE 20
#define UDP_HEADER_SIZE 8
#define RTP_HEADER_SIZE 12
#define FEC_HEADER_SIZE 16
#define MAX_SEQ_RTP 65535
#define LMAX 40
#define DxL_MAX 400
#define        MAX_RTP_SIZE        (MTU-IP_HEADER_SIZE-UDP_HEADER_SIZE-
RTP_HEADER_SIZE)

#define writes(f,x) write((f),(x),strlen(x))

   /* Declaraicion de estructuras para cabecera RTP y socket*/
  struct rtpheader hdr_fec;
  struct sockaddr_in sOut_fec;
  int socketOut_fec;

//Cabecera del paquete FEC
struct fec_header{

  unsigned int TS_recovery;
  unsigned int lenght_recovery:16;
  unsigned int SN_base_low:16;
  unsigned int mask:24;
  unsigned int E:1;
  unsigned int pt_recovery:7;
  unsigned int N:1;
  unsigned int D:1;
  unsigned int type:3;
  unsigned int index:3;
  unsigned int offset:16;
  unsigned int NA:16;
  unsigned int SN_base_ext:16;

  } fec_hdr;

void dumprtp(int socket) {
  char* buf; //buffer de recepcion
  struct rtpheader rh;  // Declaracion de la cabecera RTP "rh"
  int lengthData; //longitud de datos
  unsigned short seq=0; //numeros de secuencia
  int flag=0; //Bandera
```

```
  //Variables para operaciones FEC
  int smpte = 1;
  int cont_column = 0;
  int cont_row = 0;
  //Numero de columnas y filas -Yo lo he fijado-
  int row=5;
  int col=8;

  int pkt_fec = 0;
  int xor_ready = 0;
  int cont_fec = 0;
  int SN_first;

  //Elementos SMPTE XOR

  unsigned char row_fec [col*(8+1316)]; // Si col es 20 ROW_FEC
[26500]
  unsigned char codeword_FEC [col*(8+1316)];
  unsigned char codeword_FEC2D[row*(8+1324)];
  unsigned char pkt_empty [1332];
  unsigned char payload_FEC [(1316+16)];
  unsigned char pkt_for_XOR [1316+8];//Deberian ser 7 bytes y 6 bits,
pero vamos a poner 8 bytes
  //Vector para realizar el 2-D
  unsigned char row_fec2D[1324];

  unsigned char* payl;

  //Punteros creados para hacer el pasar a bytes de red
  unsigned int intP;
  char* charP = (char*) &intP;

  char byte1,byte2,byte3,byte4;
  unsigned int lenght_rtp;

    //Datos para el socket de envío de los paquetes FEC
    int socketOut_fec;
    int socketOut_fec2;
    char *ipOut;
    char *ipOut2;
    int portOut2;
    int portOut;
    ipOut = "224.0.1.2";
    ipOut2 = "224.0.1.2";
    portOut = 5006;
    portOut2 =5006;
    int ttl;
    struct sockaddr_in sOut_fec;
    struct sockaddr_in sOut_fec2;

    socketOut_fec = makesocket(ipOut,portOut,2,&sOut_fec);
    socketOut_fec2 = makesocket(ipOut2,portOut2,2,&sOut_fec2);

  while(1) {
    getrtp2(socket,&rh, &buf,&lengthData);

  /*  if (flag==0) { seq=rh.b.sequence; flag=1; }
    if (seq!=rh.b.sequence) {
      fprintf(stderr,"rtptsaudio: NETWORK CONGESTION - expected %d,
received %d\n",seq,rh.b.sequence);
```

```
      seq=rh.b.sequence;
    }

*/
    if(smpte == 1 )
  {
    initrtp(&hdr_fec,96/*potrebbe essere dinamico*/,RTP); //Inicializa
la cabecera RTP del paquete FEC que vamos a crear
    hdr_fec.ssrc = 0; //ssrc lo ponemos a 0

   //Inicializar la cabecera FEC

    fec_hdr.TS_recovery = 0;
    fec_hdr.lenght_recovery = 0;
    fec_hdr.SN_base_low = 0;
    fec_hdr.mask = 0;
    fec_hdr.E = 1; //RFC2733 for the FEC header
    fec_hdr.pt_recovery = 0;
    fec_hdr.N = 0;
    fec_hdr.D = 0;
    fec_hdr.type = 0;
    fec_hdr.index = 0;
    fec_hdr.offset = col;
    fec_hdr.NA = row;
   fec_hdr.SN_base_ext = 0;

    //fcntl (socketOut_fec, F_SETFL, O_NONBLOCK);

    int j, lenght_rtp;

    //Inicializar todas las matrices  a 0
    for(j=0;j<col*1324;j++) codeword_FEC[j] = 0;
    for(j=0;j<1324;j++) pkt_for_XOR [j] = 0;
    for(j=0;j<1332;j++) pkt_empty [j] = 0;

    lenght_rtp = 1316; /*Debe de ser la suma de la lista CSRC,
éxtension de cabecera, carga útil de RTP y el relleno */
  }

    int i,j;
    if(smpte==1)

         {
            if ((cont_row==0) && (cont_column==0)) {  //la primera vez
entra porque los contadores se inicializan en 0
              SN_first = rh.b.sequence; //Para marcar el primer
paquete
              hdr_fec.timestamp = rh.timestamp; //Para guardar el
tiempo del primer paquete en la cabecera RTP del paquete FEC
            }
            /*Hacer la cadena de bits de XOR*/

            for(i=0;i<1324;i++) {
                pkt_for_XOR[i] = 0;
                row_fec2D[i] = 0;
                } //Ponemos 0 en todas las casillas

            //Metemos la cabecera del paquete RTP que hemos recibido
en un vector para hacer la operacion XOR
            pkt_for_XOR[0] |= ((((char) rh.b.p)<<7)&0x80); //1000 0000
Padding
```

```
            pkt_for_XOR[0] |= ((((char) rh.b.x)<<6)&0x40); //0100 0000
extension
            pkt_for_XOR[0] |= ((((char) rh.b.cc)<<2)&0x3c);//0011 1100
CSRC
            pkt_for_XOR[0] |= ((((char) rh.b.m)<<1)&0x02); //0000 0010
Marcador
            pkt_for_XOR[0] |= ((((char) rh.b.pt)<<2)&0x01);//0000 0001
tipo de carga util
            pkt_for_XOR[1] |= ((((char) rh.b.pt)<<2)&0xfc);//1111 1100
            //Convertimos los datos que hemos recibido en la cabecera
RTP y lo guardamos en el paquete para operacion XOR
            intP = htonl(rh.timestamp);
            memcpy(&byte1,charP,1); //Time stamp

            pkt_for_XOR[1] |= ((((char) byte1)<<2)&0x03);//0000 0011
con esta mascara asegura que lo que haya es lo correcto m[]|= 0000
0000
            pkt_for_XOR[2] |= ((((char) byte1)<<2)&0xfc);//1111 1100
            memcpy(&byte2,charP+1,1);
            pkt_for_XOR[2] |= ((((char) byte2)<<2)&0x03);//0000 0011
            pkt_for_XOR[3] |= ((((char) byte2)<<2)&0xfc);//1111 1100
            memcpy(&byte3,charP+2,1);
            pkt_for_XOR[3] |= ((((char) byte3)<<2)&0x03);//0000 0011
            pkt_for_XOR[4] |= ((((char) byte3)<<2)&0xfc);//1111 1100
            memcpy(&byte4,charP+3,1);
            pkt_for_XOR[4] |= ((((char) byte4)<<2)&0x03);//0000 0011
            pkt_for_XOR[5] |= ((((char) byte4)<<2)&0xfc);//1111 1100

            intP = htonl(lenght_rtp);
            memcpy(&byte1,charP+2,1);
            pkt_for_XOR[5] |= ((((char) byte1)<<2)&0x03);//0000 0011
            pkt_for_XOR[6] |= ((((char) byte1)<<2)&0xfc);//1111 0011
            memcpy(&byte2,charP+3,1);
            pkt_for_XOR[6] |= ((((char) byte2)<<2)&0x03);//0000 0011
            pkt_for_XOR[7] |= ((((char) byte2)<<2)&0xfc);//1111 0011

            //Guardamos los datos utiles

            for (j = 0; j < 1316; j++){      //Tomo como valor de la
bits restantes de 1316, el tamñao de la carga util del paquete de la
fuente RTP
              pkt_for_XOR[7+j] |= ((((char) buf[j])<<2)&0x03);//0000
0011 //Con esto siempre dejara lo que haya sin alterar
              pkt_for_XOR[8+j] |= ((((char) buf[j])<<2)&0xfc);//1111
0011
            }
            //Guardamos los bits que hemos ido almacenando en
PKT_FOR_XOR en row fec, vamos contando hasta el maximo de columnas de
la matriz--YO HE FORZADO 5--
            for (j = (0+1324*cont_column); j < (1324 +
1324*cont_column); j++) //el primer caso es j=0, j<1324
            {
                row_fec[j] = pkt_for_XOR[j-(1324*cont_column)];//(J-
(1324*cont_column == Para empezar des de 0 siempre.
            }
            cont_column++;

            for (j=0; j<1324; j++) row_fec2D[j]^= pkt_for_XOR[j];

            if (cont_column==col)
            {
```

```
              for(j=0;j<col*1324;j++)
              {
                  //Operacion OR exclusivo si 1^1 = 0 0^0 = 0 en caso
contrario 1
                  codeword_FEC[j]^=row_fec[j];//Òperacion  XOR  cuando
llegamos al maximo de paquetes-YO HE FORZADO 5
              }

              for (j=(0+1324*cont_row); j<(1324+1324*cont_row); j++)
              {
                codeword_FEC2D[j] = row_fec2D[j-(1324*cont_row)];
              }

            cont_column = 0; //Ponemos el contador de columnas porque
hemos llegado al maximo
             //AUmentamos el contador de filas para volver a comenzar
el proceso
            cont_row++;
             }



              if(cont_row == row)
              {

               while(cont_fec!=row)
               {


               for(i=0;i<1324;i++) payload_FEC[i]=0;
               hdr_fec.b.p          |=          (unsigned          int)
(((codeword_FEC2D[0+cont_fec*1324])>>7)&0x01); //0000 0001
               hdr_fec.b.x          |=          (unsigned          int)
(((codeword_FEC2D[0+cont_fec*1324])>>6)&0x01); //0000 0001
               hdr_fec.b.cc          |=          (unsigned          int)
(((codeword_FEC2D[0+cont_fec*1324])>>2)&0x0f);//0000 1111
               hdr_fec.b.m          |=          (unsigned          int)
(((codeword_FEC2D[0+cont_fec*1324])>>1)&0x01); //0000 00001
               fec_hdr.pt_recovery        |=        (unsigned        int)
(((codeword_FEC2D[0+cont_fec*1324])<<6)&0x80); //1000  0000   El ultimo
bit del primer byte es el septimo bit del pt_recovery
               fec_hdr.pt_recovery        |=        (unsigned        int)
(((codeword_FEC2D[1+cont_fec*1324])>>2)&0x3f); //0011 1111 Y ahora los
seis siguientes
               intP = 0;
               byte1 = 0;
               byte2 = 0;
               byte3 = 0;
               byte4 = 0;
               byte1 |= (((codeword_FEC2D[1+cont_fec*1324])>>2)&0xc0);
               byte1 |= (((codeword_FEC2D[2+cont_fec*1324])>>2)&0x3f);
               memcpy(&byte1,charP,1);
               byte2 |= (((codeword_FEC2D[2+cont_fec*1324])>>2)&0xc0);
               byte2 |= (((codeword_FEC2D[3+cont_fec*1324])>>2)&0x3f);
               memcpy(&byte2,charP+1,1);
               byte3 |= (((codeword_FEC2D[3+cont_fec*1324])>>2)&0xc0);
               byte3 |= (((codeword_FEC2D[4+cont_fec*1324])>>2)&0x3f);
               memcpy(&byte3,charP+2,1);
               byte4 |= (((codeword_FEC2D[4+cont_fec*1324])>>2)&0xc0);
               byte4 |= (((codeword_FEC2D[5+cont_fec*1324])>>2)&0x3f);
```

```
                memcpy(&byte4,charP+3,1);
                fec_hdr.TS_recovery = ntohl(intP);
                intP = 0;
                byte1 = 0;
                byte2 = 0;
                byte1 |= (((codeword_FEC2D[5+cont_fec*1324])>>2)&0xc0);
                byte1 |= (((codeword_FEC2D[6+cont_fec*1324])>>2)&0x3f);
                memcpy(&byte1,charP+2,1);
                byte2 |= (((codeword_FEC2D[6+cont_fec*1324])>>2)&0xc0);
                byte2 |= (((codeword_FEC2D[7+cont_fec*1324])>>2)&0x3f);
                memcpy(&byte2,charP+3,1);
                fec_hdr.lenght_recovery = ntohl(intP);

                // packetizer_FEC_header, Siguiendo lo que esta escrito
en el standard

                intP = htonl(fec_hdr.SN_base_low);
                memcpy(&payload_FEC [0],charP+2,2);
                intP = htonl(fec_hdr.lenght_recovery);
                memcpy(&payload_FEC [2],charP+2,2);
                payload_FEC [4] |= ((((char)fec_hdr.E)<<7)&0x80);
                payload_FEC                    [4]                    |=
((((char)fec_hdr.pt_recovery)>>0)&0x7f);
                intP = htonl(fec_hdr.mask);
                memcpy(&payload_FEC [5],charP+1,3);
                intP = htonl(fec_hdr.TS_recovery);
                memcpy(&payload_FEC [8],&intP,4);
                payload_FEC [12] |= ((((char)fec_hdr.N)<<7)&0x80);
                payload_FEC [12] |= ((((char)fec_hdr.D)<<6)&0x40);
                payload_FEC [12] |= ((((char)fec_hdr.type)<<3)&0x38);
                payload_FEC [12] |= ((((char)fec_hdr.index)<<0)&0x07);
                intP = htonl(fec_hdr.offset);
                memcpy(&payload_FEC [13],charP+3,1);
                intP = htonl(fec_hdr.NA);
                memcpy(&payload_FEC [14],charP+3,1);
                intP = htonl(fec_hdr.SN_base_ext);
                memcpy(&payload_FEC [15],charP+3,1);

                for(i=0;i<1316;i++)       //El  indice  del  payload_FEC
tiene que comenzar en  17, mientras que las palabras de codigo en 8
                {
                   payload_FEC      [i+16]      |=      (((codeword_FEC2D
[i+7+cont_fec*1324])>>2)&0xc0);
                   payload_FEC      [i+16]      |=      (((codeword_FEC2D
[i+8+cont_fec*1324])>>2)&0x3f);

                }
                //ENVIAMOS  El  paquete  de  reparacion,  su  carga  util  +
cabecera 1316 + 16
                sendrtp2(socketOut_fec2,&sOut_fec2,&hdr_fec,payload_FEC
,1332);

                fec_hdr.SN_base_low  ++;  //DE  esta  manera  tenemos  la
SN_base_low para cada paquete FEC
                cont_fec++;

                 xor_ready = 1;
                 cont_row = 0;

                 fec_hdr.SN_base_low = SN_first;
               }
```

```
                cont_fec = 0;
            }

                //EN este momento hemos rellenado FIlas x Columnas
Paquetes RTP el vector
                //A continuacion vamos a hacer los paquetes
                while (xor_ready==1)
                {
                    printf("Entra 1D");
                   for(i=0;i<1324;i++) payload_FEC[i]=0;
                   hdr_fec.b.p             |=          (unsigned          int)
(((codeword_FEC[0+cont_fec*1324])>>7)&0x01); //0000 0001
                   hdr_fec.b.x             |=          (unsigned          int)
(((codeword_FEC[0+cont_fec*1324])>>6)&0x01); //0000 0001
                   hdr_fec.b.cc            |=          (unsigned          int)
(((codeword_FEC[0+cont_fec*1324])>>2)&0x0f);//0000 1111
                   hdr_fec.b.m             |=          (unsigned          int)
(((codeword_FEC[0+cont_fec*1324])>>1)&0x01); //0000 00001
                   fec_hdr.pt_recovery       |=       (unsigned       int)
(((codeword_FEC[0+cont_fec*1324])<<6)&0x80);  //1000  0000   El  ultimo
bit del primer byte es el septimo bit del pt_recovery
                   fec_hdr.pt_recovery       |=       (unsigned       int)
(((codeword_FEC[1+cont_fec*1324])>>2)&0x3f);  //0011  1111  Y  ahora  los
seis siguientes
                   intP = 0;
                   byte1 = 0;
                   byte2 = 0;
                   byte3 = 0;
                   byte4 = 0;
                   byte1 |= (((codeword_FEC[1+cont_fec*1324])>>2)&0xc0);
                   byte1 |= (((codeword_FEC[2+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte1,charP,1);
                   byte2 |= (((codeword_FEC[2+cont_fec*1324])>>2)&0xc0);
                   byte2 |= (((codeword_FEC[3+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte2,charP+1,1);
                   byte3 |= (((codeword_FEC[3+cont_fec*1324])>>2)&0xc0);
                   byte3 |= (((codeword_FEC[4+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte3,charP+2,1);
                   byte4 |= (((codeword_FEC[4+cont_fec*1324])>>2)&0xc0);
                   byte4 |= (((codeword_FEC[5+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte4,charP+3,1);
                   fec_hdr.TS_recovery = ntohl(intP);
                   intP = 0;
                   byte1 = 0;
                   byte2 = 0;
                   byte1 |= (((codeword_FEC[5+cont_fec*1324])>>2)&0xc0);
                   byte1 |= (((codeword_FEC[6+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte1,charP+2,1);
                   byte2 |= (((codeword_FEC[6+cont_fec*1324])>>2)&0xc0);
                   byte2 |= (((codeword_FEC[7+cont_fec*1324])>>2)&0x3f);
                   memcpy(&byte2,charP+3,1);
                   fec_hdr.lenght_recovery = ntohl(intP);

                   // packetizer_FEC_header, Siguiendo lo que esta escrito
en el standard

                   intP = htonl(fec_hdr.SN_base_low);
                   memcpy(&payload_FEC [0],charP+2,2);
                   intP = htonl(fec_hdr.lenght_recovery);
                   memcpy(&payload_FEC [2],charP+2,2);
                   payload_FEC [4] |= ((((char)fec_hdr.E)<<7)&0x80);
```

```
                payload_FEC                    [4]                    |=
((((char)fec_hdr.pt_recovery)>>0)&0x7f);
                intP = htonl(fec_hdr.mask);
                memcpy(&payload_FEC [5],charP+1,3);
                intP = htonl(fec_hdr.TS_recovery);
                memcpy(&payload_FEC [8],&intP,4);
                payload_FEC [12] |= ((((char)fec_hdr.N)<<7)&0x80);
                payload_FEC [12] |= ((((char)fec_hdr.D)<<6)&0x40);
                payload_FEC [12] |= ((((char)fec_hdr.type)<<3)&0x38);
                payload_FEC [12] |= ((((char)fec_hdr.index)<<0)&0x07);
                intP = htonl(fec_hdr.offset);
                memcpy(&payload_FEC [13],charP+3,1);
                intP = htonl(fec_hdr.NA);
                memcpy(&payload_FEC [14],charP+3,1);
                intP = htonl(fec_hdr.SN_base_ext);
                memcpy(&payload_FEC [15],charP+3,1);

                for(i=0;i<1316;i++)        //El  indice  del  payload_FEC
tiene que comenzar en  17, mientras que las palabras de codigo en 8
                {
                   payload_FEC      [i+16]      |=      (((codeword_FEC
[i+7+cont_fec*1324])>>2)&0xc0);
                   payload_FEC      [i+16]      |=      (((codeword_FEC
[i+8+cont_fec*1324])>>2)&0x3f);

                }
                //ENVIAMOS  El  paquete  de  reparacion,  su  carga  util  +
cabecera 1316 + 16
                sendrtp2(socketOut_fec,&sOut_fec,&hdr_fec,payload_FEC
,1332);

                fec_hdr.SN_base_low  ++;  //DE  esta  manera  tenemos  la
SN_base_low para cada paquete FEC
                cont_fec++;

                if (cont_fec == col)
                {
                  xor_ready=0;
                  cont_fec=0;
                  for(j=0;j<col*1324;j++) codeword_FEC[j] = 0;
                }
              }
            }
    seq++;


  }
}
int main(int argc, char *argv[]) {

  struct sockaddr_in si;
  int socketIn;

  char *ip;
  int port;

  fprintf(stderr,"Rtp dump\n");

  if (argc == 1) {
    ip   = "224.0.1.2";
    port = 5004;
```

```
  }
  else if (argc == 3) {
    ip   = argv[1];
    port = atoi(argv[2]);
  }
  else {
    fprintf(stderr,"Usage %s ip port\n",argv[0]);
    exit(1);
  }

  fprintf(stderr,"Using %s:%d\n",ip,port);

  socketIn  = makeclientsocket(ip,port,2/*ttl*/,&si);
  dumprtp(socketIn);

  close(socketIn);
  return(0);
}
```

Funciones utilizadas en el servidor AL-FEC

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "rtp_modified.h"

Estructura de la cabecera de un paquete FEC

struct fec_header {

  unsigned int TS_recovery;
  unsigned int lenght_recovery:16;
  unsigned int SN_base_low:16;
  unsigned int mask:24;
  unsigned int E:1;
  unsigned int pt_recovery:7;
  unsigned int N:1;
  unsigned int D:1;
  unsigned int type:3;
  unsigned int index:3;
  unsigned int offset:16;
  unsigned int NA:16;
  unsigned int SN_base_ext:16;

};
```

Función que coge los datos de los paquetes RTP y prepara un paquete para mas adelante realizar la función XOR.

```
int pkt_for_XOR_building (unsigned char *data_rtp, unsigned char
*pkt_for_XOR , struct rtpheader *rh, unsigned int lenght_rtp) {

 int j;
 unsigned int intP;
 char* charP = (char*) &intP;
 unsigned char byte1,byte2,byte3,byte4;

 for(j=0;j<lenght_rtp+8;j++) pkt_for_XOR[j]=0;

 pkt_for_XOR[0] |= ((((unsigned char) rh->b.p)<<7)&0x80);
 pkt_for_XOR[0] |= ((((unsigned char) rh->b.x)<<6)&0x40);
 pkt_for_XOR[0] |= ((((unsigned char) rh->b.cc)<<2)&0x3c);
 pkt_for_XOR[0] |= ((((unsigned char) rh->b.m)<<1)&0x02);
 pkt_for_XOR[0] |= ((((unsigned char) rh->b.pt)<<2)&0x01);
 pkt_for_XOR[1] |= ((((unsigned char) rh->b.pt)<<2)&0xfc);
 intP = htonl(rh->timestamp);
 memcpy(&byte1, charP,1); //first byte of intP ( i.e. the one containg
the most important bits)
 pkt_for_XOR[1] |= ((((unsigned char) byte1)>>6)&0x03);
 pkt_for_XOR[2] |= ((((unsigned char) byte1)<<2)&0xfc);
 memcpy(&byte2, charP+1,1);
 pkt_for_XOR[2] |= ((((unsigned char) byte2)>>6)&0x03);
 pkt_for_XOR[3] |= ((((unsigned char) byte2)<<2)&0xfc);
 memcpy(&byte3,charP+2,1);
 pkt_for_XOR[3] |= ((((unsigned char) byte3)>>6)&0x03);
 pkt_for_XOR[4] |= ((((unsigned char) byte3)<<2)&0xfc);
 memcpy(&byte4,charP+3,1);
 pkt_for_XOR[4] |= ((((unsigned char) byte4)>>6)&0x03);
 pkt_for_XOR[5] |= ((((unsigned char) byte4)<<2)&0xfc);
 intP = htonl(lenght_rtp);
 memcpy(&byte1, charP+2,1);
 pkt_for_XOR[5] |= ((((unsigned char) byte1)>>6)&0x03);
 pkt_for_XOR[6] |= ((((unsigned char) byte1)<<2)&0xfc);
 memcpy(&byte2, charP+3,1);
 pkt_for_XOR[6] |= ((((unsigned char) byte2)>>6)&0x03);
 pkt_for_XOR[7] |= ((((unsigned char) byte2)<<2)&0xfc);

 for (j = 0; j < lenght_rtp; j++){       /*I take as value of the
remaining bit 1316, being the size of the RTP source packets'
payload*/

          pkt_for_XOR[7+j]        |=        ((((unsigned        char)
data_rtp[j])>>6)&0x03);
          pkt_for_XOR[8+j]        |=        ((((unsigned        char)
data_rtp[j])<<2)&0xfc);
 }
//fprintf(stderr,"%d\n",pkt_for_XOR[1004]);
return 1;
}
```

Función para iniciacializar la cabecera de un paquete RTP

```
void initrtp(struct rtpheader *foo,int pt, int type) { /* fill in the
  foo->b.v=2;
  foo->b.p=0;
  foo->b.x=0;
  foo->b.cc=0;
  foo->b.m=0;
  foo->b.pt=pt;
  foo->b.sequence=rand() & 65535;
  foo->timestamp=rand();
  foo->ssrc=rand();
  foo->type = type;
}
```

Función para recuperar los paquetes RTP para más tarde crear los paquetes FEC.

```
int getrtp2(int fd, struct rtpheader *rh, char* data, int* lengthData)
{
  static char buf[1600];
  char *pointer = buf;
  unsigned int intP;
  char* charP = (char*) &intP;
  int headerSize;
  int lengthPacket;
  lengthPacket=recv(fd,buf,1328,0);
  if (lengthPacket==0)
    exit(1);
 if (lengthPacket<0) {
    //fprintf(stderr,"socket read error\n");
    return 0;
  }

  rh->b.v  = (unsigned int) ((buf[0]>>6)&0x03);
  rh->b.p  = (unsigned int) ((buf[0]>>5)&0x01);
  rh->b.x  = (unsigned int) ((buf[0]>>4)&0x01);
  rh->b.cc = (unsigned int) ((buf[0]>>0)&0x0f);
  rh->b.m  = (unsigned int) ((buf[1]>>7)&0x01);
  rh->b.pt = (unsigned int) ((buf[1]>>0)&0x7f);
  intP = 0;
  memcpy(charP+2,&buf[2],2);
  rh->b.sequence = ntohl(intP);
  intP = 0;
  memcpy(charP,&buf[4],4);
  rh->timestamp = ntohl(intP);

  headerSize = 12 + 4*rh->b.cc; /* in bytes */

  *lengthData = lengthPacket - headerSize;
   memcpy(data, pointer + headerSize,lengthPacket - headerSize);
  //*data = (char*) buf + headerSize;

  //   fprintf(stderr,"Reading rtp: v=%x p=%x x=%x cc=%x m=%x pt=%x
seq=%x  ts=%x  lgth=%d\n",rh->b.v,rh->b.p,rh->b.x,rh->b.cc,rh->b.m,rh-
>b.pt,rh->b.sequence,rh->timestamp,lengthPacket);

  return 1;
}
```

Función para el envío de paquetes FEC a la dirección multicast.

```
int sendrtp2(int fd, struct sockaddr_in *sSockAddr, struct rtpheader
*foo, char *data, int len) {
  char *buf;
  unsigned int intP;
  char* charP = (char*) &intP;//serve per memorizzare il sequence
number: infatti ho bisogno di un puntatore
  int headerSize;
  if(foo->type == RTP) {
  buf=(char*)alloca(len+72);
  buf[0]  = 0x00;
  buf[0] |= ((((char) foo->b.v)<<6)&0xc0);
  buf[0] |= ((((char) foo->b.p)<<5)&0x20);
  buf[0] |= ((((char) foo->b.x)<<4)&0x10);
  buf[0] |= ((((char) foo->b.cc)<<0)&0x0f);
  buf[1]  = 0x00;
  buf[1] |= ((((char) foo->b.m)<<7)&0x80);
  buf[1] |= ((((char) foo->b.pt)<<0)&0x7f);
  intP = htonl(foo->b.sequence);
  memcpy(&buf[2],charP+2,2);
  intP = htonl(foo->timestamp);
  memcpy(&buf[4],&intP,4);
  /* SSRC: not implemented */
  buf[8]  = 0x0f;
  buf[9]  = 0x0f;
  buf[10] = 0x0f;
  buf[11] = 0x0f;
  headerSize = 12 + 4*foo->b.cc; /* in bytes */
  memcpy(buf+headerSize,data,len);

  //   fprintf(stderr,"Sending rtp: v=%x p=%x x=%x cc=%x m=%x pt=%x
seq=%x    ts=%x    lgth=%d\n",foo->b.v,foo->b.p,foo->b.x,foo->b.cc,foo-
>b.m,foo->b.pt,foo->b.sequence,foo->timestamp,len+headerSize);

  foo->b.sequence++;
  } else {  //UDP
    buf = data;
    headerSize = 0;
  }
  return        sendto(fd,buf,len+headerSize,0,(struct        sockaddr
*)sSockAddr,sizeof(*sSockAddr));
}
```

Función para crear un socket multicast.

```
int makeclientsocket(char *szAddr,unsigned short port,int TTL,struct
sockaddr_in *sSockAddr) {
  int socket=makesocket(szAddr,port,TTL,sSockAddr);
  struct ip_mreq blub;
  struct sockaddr_in sin;
  unsigned int tempaddr;
  sin.sin_family=AF_INET;
  sin.sin_port=htons(port);
  sin.sin_addr.s_addr=inet_addr(szAddr);
  if (bind(socket,(struct sockaddr *)&sin,sizeof(sin))) {
    perror("bind failed");
    exit(1);
```

```
  }
  tempaddr=inet_addr(szAddr);
  if ((ntohl(tempaddr) >> 28) == 0xe) {
    blub.imr_multiaddr.s_addr = inet_addr(szAddr);
    blub.imr_interface.s_addr = 0;
    if
(setsockopt(socket,IPPROTO_IP,IP_ADD_MEMBERSHIP,&blub,sizeof(blub))) {
      perror("setsockopt     IP_ADD_MEMBERSHIP     failed     (multicast
kernel?)");
      exit(1);
    }
  }
  return socket;
}
```

Biblioteca

Biblioteca con las cabeceras de todas las funciones utilizadas y las diferentes estructuras para los paquetes RTP y RTCP Feedback

```
#ifndef _RTP_H
#define _RTP_H

#include <sys/socket.h>

enum {RTP_PS,RTP_TS,RTP_NONE,MAP_TS};
enum {RTP, UDP};

struct rtpbits {
  unsigned int v:2;           /* version: 2 */
  unsigned int p:1;           /* is there padding appended: 0 */
  unsigned int x:1;           /* number of extension headers: 0 */
  unsigned int cc:4;          /* number of CSRC identifiers: 0 */
  unsigned int m:1;           /* marker: 0 */
  unsigned int pt:7;          /* payload type: 33 for MPEG2 TS - RFC
1890 */
  unsigned int sequence:16;   /* sequence number: random */
};
struct packet_RTCPFB {      /* nonflash player ubuntu-compound RTCP
pckt implemented */ //ricordare:stesso socket ma num. porta +1 !!
  unsigned int v:2;
  unsigned int p:1;
  unsigned int FMT:5;       /* Feedack Message Type (must be 1 for
transport layer FB msg) */
  unsigned int PT:8;        /* fixed value 205 (means transport layer
feedback message) */
  unsigned int length:16;   /* length of the packet */
  unsigned int ssrc_psender:32; /* synchronization source of the
packet sender for the client (always the same of original packet) */
  unsigned int ssrc_msource:32; /* synchronization source of media
source for the client (always the same of original packet) */
  unsigned int PID:16;      /* Packet ID = sequence number of the
packet lost */
  unsigned int BLP:16;      /* Bitmask of the following lost packet
(not implemented) */
};

struct rtpheader {  /* in network byte order */
```

```
  struct rtpbits b;
  int timestamp;    /* start: random */
  int ssrc;     /* random */
  int type;     /* RTP or UDP */
};
struct rtpheaderRET {   /* in network byte order */
  struct rtpbits b;
  int timestamp;    /* start: random */
  int ssrc;     /* random */
  int type;     /* RTP or UDP */
};


void initrtp(struct rtpheader *foo,int pt, int type); /* fill in the
MPEG-2 TS deefaults */
int sendrtp(int fd, struct sockaddr_in *sSockAddr, struct rtpheader
*foo, char *data, int len);
int getrtp2(unsigned char** datos[], int fd, struct rtpheader *rh,
char** data, int* lengthData);
int sendrtp2(int fd, struct sockaddr_in *sSockAddr, struct rtpheader
*foo, char** data, int len);
int getrtp(int fd, struct rtpheader *rh, char** data, int*
lengthData);
int makesocket(char *szAddr,unsigned short port,int TTL,struct
sockaddr_in *sSockAddr);
int makeclientsocket(char *szAddr,unsigned short port,int TTL,struct
sockaddr_in *sSockAddr);
int pkt_for_XOR_building (unsigned char *data_rtp, unsigned char
*pkt_for_XOR , struct rtpheader *rh, unsigned int lenght_rtp)
int secuencia(char buf[2])
#endif
```